

---

---

# Object-Oriented Modeling and Coordination of Mobile Robots

---

---

By

ANDREAS SCHIERL

Department of Software Engineering  
UNIVERSITY OF AUGSBURG

A dissertation submitted to the University of Augsburg in  
accordance with the requirements of the degree of DOCTOR  
OF NATURAL SCIENCES in the Faculty of Applied Computer  
Science.

MARCH 2016

Supervisor: Prof. Dr. Wolfgang Reif  
Advisor: Prof. Dr. Alexander Knapp  
Defense: May 30, 2016

---

## Abstract

Nowadays, industrial robots play an important role automating recurring manufacturing tasks. New trends towards *Smart Factory* and *Industry 4.0* however take a more product-driven approach and demand for more flexibility of the robotic systems. When a varying order of processing steps is required, intra-factory logistics has to cope with the new challenges. To achieve this flexibility, mobile robots can be used for transporting goods, or even mobile manipulators consisting of a mobile platform and a robot arm for independently grasping work pieces and manipulating them while in motion. Working with mobile robots however poses new challenges that did not yet occur for industrial manipulators: First, mobile robots have a greater position inaccuracy and typically work in not fully structured environments, requiring to interpret sensor data and to more often react to events from the environment. Furthermore, independent mobile robots introduce the aspect of distribution. For mobile manipulators, an additional challenge arises from the combination of platform and arm, where platform and arm, but also sensors have to be coordinated to achieve the desired behavior.

The main contribution of this work is an approach that allows the *object-oriented modeling and coordination of mobile robots*, supporting the cooperation of mobile manipulators. Within a mobile manipulator, the approach allows to define real-time reactions to sensor data and to synchronize the different actuators and sensors present, allowing sensor-aware combinations of motions for platform and arm. Moreover, the approach facilitates an easy way of programming, provides means to handle kinematic restrictions or redundancy, and supports advanced capabilities such as impedance control to mitigate position uncertainty. Working with multiple independent mobile robots, each has a different knowledge about its environment, based on the available sensors. These different views are modeled, allowing consistent coordination of robots in applications using the data available on each robot. To cope with geometric uncertainty, sensors are modeled and the relationship between their measurements and geometric aspects is defined. Based on these definitions and incoming sensor data, position estimates are automatically derived. Additionally, the more dynamic environment leads to different possible outcomes of task execution. These are explicitly modeled and can be used to define reactive behavior. The approach was successfully evaluated based on two application examples, ranging from physical interaction between two mobile manipulators handing over a work-piece to gesture control of a quadcopter for carrying goods.



## ACKNOWLEDGEMENTS

This thesis originates from my work as a researcher at the Institute for Software and Systems Engineering at the University of Augsburg. Thus, I want to thank everyone who supported my work there, starting with Prof. Dr. Wolfgang Reif for supervising this work, who offered me the opportunity to work in the interesting and cost-intensive field of mobile robotics and supported me with interesting discussions, as well as Prof. Dr. Alexander Knapp as second examiner. This work would not have been possible without the preparatory work of my colleagues. Thus I especially want to thank Dr. Alwin Hoffmann, Dr. Andreas Angerer and Dr. Michael Vistein for their contributions concerning object-oriented robot programming, as well as for many inspiring discussions and for proofreading this work and providing valuable feedback. Completing the robotics group, Ludwig Nägele, Miroslav Macho, Constantin Wanninger and Alexander Poeppel have contributed through discussions and their support.

Moreover, I'd like to thank the organizers of the *BRICS research camps* on mobile manipulation aspects (organized by the European BRICS project) for providing me with a broader view on mobile manipulation through talks and interesting discussions with the further participants. A special thank goes to Prof. Dr. Herman Bruyninckx for the helpful discussions and his input concerning robot structure and capabilities in general and the possibilities and restrictions of mobile manipulators.

Finally, my thank goes to my family for unconditionally supporting and motivating me during the work on my thesis, as well as to my friends for their support and understanding.

## TABLE OF CONTENTS

<b>Table of Contents</b>	<b>iv</b>
<b>1 Introduction</b>	<b>1</b>
<b>2 Object-Oriented Robot Programming and Mobile Robots</b>	<b>7</b>
2.1 The Robotics API – Object-oriented Programming of Industrial Robots . . . . .	9
2.1.1 Specification and Execution of Real-Time Critical Tasks . . . . .	10
2.1.2 Object-oriented Application Development for Industrial Robots . . . . .	16
2.2 Using Object-Oriented Programming for Mobile Robots . . . . .	18
2.3 Software Structure and Distribution for Mobile Robots . . . . .	19
2.3.1 Control and Deterministic Reactions – the Real-Time Level . . . . .	21
2.3.2 Freely Communicating Components – the System Level . . . . .	24
2.3.3 Coordinating What Happens – the Application Level . . . . .	26
<b>3 Case Study: Cooperating Mobile Robots</b>	<b>29</b>
3.1 Influencing the Environment – Actuators . . . . .	30
3.1.1 Mobile Manipulator – KUKA youBot . . . . .	30
3.1.2 Flying Robot – r0sewhite Saphira Quadcopter . . . . .	32
3.2 Observing the Environment – Sensors . . . . .	33
3.2.1 Onboard Sensing – Hokuyo URG-04LX-UG01 Laser Scanner . . . . .	33
3.2.2 External Optical Tracking – Vicon T-Series . . . . .	34
3.3 Application Examples . . . . .	35
3.3.1 Handover in Motion . . . . .	35
3.3.2 Gesture Control of a Quadcopter . . . . .	36
<b>4 Physical Objects and Application Geometry</b>	<b>39</b>
4.1 Representing Cartesian Space . . . . .	41
4.1.1 Mathematical Representation – Vectors, Rotations and Matrices . . . . .	41
4.1.2 Frame and Relation – Named Places and their Relationships . . . . .	44
4.1.3 Point and Orientation – Positions and Coordinate Axes . . . . .	47
4.1.4 Pose and FramePose – Positions with Orientation . . . . .	48

4.1.5	Velocity and FrameVelocity – Talking about Motion . . . . .	51
4.2	Modeling Physical Objects and their Geometric Features . . . . .	55
4.2.1	Mobile Manipulator – youBot Platform, Arm and Gripper . . . . .	56
4.2.2	Quadcopter – r0sewhite Saphira with Autoquad Flight Controller . . . . .	61
4.2.3	Local Sensing – Hokuyo Laser Scanner . . . . .	61
4.2.4	Global Position Reference – Vicon Tracking System . . . . .	62
4.3	Setting Up the Environment . . . . .	62
4.4	Comparison with Former Work on the Robotics API . . . . .	64
4.5	Related Work . . . . .	65
<b>5</b>	<b>Motions and Tool Actions</b>	<b>69</b>
5.1	Actions to Describe Motions and Tool Operations . . . . .	70
5.1.1	Motions in Cartesian Space . . . . .	72
5.1.2	Motions in Joint Space . . . . .	74
5.1.3	Tool Operations and State Transitions . . . . .	75
5.2	Device Parameters to Configure Motions . . . . .	76
5.2.1	Cartesian Limit and Joint Limit – Limiting Motion Path and Speed . . . . .	76
5.2.2	Controller Parameter – Controller Choice for Motion Execution . . . . .	77
5.2.3	Motion Center – Specifying what Moves for more Flexible Motions . . . . .	78
5.2.4	Frame Projectors – Working with Kinematic Restrictions and Redundancy . . . . .	79
5.3	Comparison with Previous Work on the Robotics API . . . . .	85
5.4	Related Work . . . . .	85
<b>6</b>	<b>Real-time Commands and Execution</b>	<b>89</b>
6.1	Commands Linking Actions to Actuators . . . . .	90
6.1.1	Realtime Values – Working with Time-Variable Data . . . . .	90
6.1.2	Commands – Executable Specifications for Actuators . . . . .	92
6.1.3	Command Operations and Scheduling Rules – Controlling Commands . . . . .	94
6.2	Executing Actions – From Commands to Real-Time Control . . . . .	96
6.2.1	Making Actions Executable . . . . .	97
6.2.2	Converting Commands into Data-Flow Graphs . . . . .	100
6.2.3	Handling Life-Cycle and Scheduled Operations . . . . .	103
6.3	Comparison with Previous Work on the Robotics API . . . . .	104
6.4	Related Work . . . . .	107
<b>7</b>	<b>Device Capabilities and Synchronization</b>	<b>111</b>
7.1	Levels of Robot Synchronization . . . . .	112
7.1.1	Workflow Synchronization . . . . .	113
7.1.2	Time Synchronization . . . . .	114

## TABLE OF CONTENTS

---

7.1.3	Data Synchronization . . . . .	114
7.2	Modeling Device Capabilities . . . . .	115
7.2.1	DeviceInterfaces and Activities – Accessing Device Capabilities . . . . .	115
7.2.2	Implementing and Executing Activities through Commands . . . . .	118
7.3	Composed Activities – Combining Capabilities . . . . .	121
7.3.1	Parallel and Sequential – Composition with Real-Time Guarantees . . . . .	122
7.3.2	ThreadActivity – Encapsulating Long Sequences . . . . .	124
7.3.3	StateChartActivity – Encapsulating Reactive Behavior . . . . .	126
7.4	Comparison with Previous Work on the Robotics API . . . . .	128
7.5	Related Work . . . . .	130
<b>8</b>	<b>Sensors and Observations</b>	<b>133</b>
8.1	Accessing Sensor Data . . . . .	134
8.1.1	Accessing Individual Sensors . . . . .	134
8.1.2	Consistent Combination of Sensor Data . . . . .	136
8.1.3	Sensors in the Case Study . . . . .	138
8.2	Integrating Sensor Data into the World Model . . . . .	140
8.2.1	Observations – Describing what is Measured . . . . .	140
8.2.2	Estimators – Using Sensor Data to Update the World Model . . . . .	143
8.3	Comparison with Previous Work on the Robotics API . . . . .	147
8.4	Related Work . . . . .	148
<b>9</b>	<b>Execution Environments and Deployment</b>	<b>151</b>
9.1	Deployment and Distribution for Mobile Robots . . . . .	152
9.1.1	Connecting Devices to Execution Environments . . . . .	152
9.1.2	Working with Multiple Execution Environments . . . . .	156
9.2	Execution Environment Implementations . . . . .	158
9.2.1	The Java Control Core – for Debugging and Simulation . . . . .	158
9.2.2	The SoftRobot Robot Control Core – with Real-Time Guarantees . . . . .	160
9.3	Comparison with Previous Work on the Robotics API . . . . .	165
9.4	Related Work . . . . .	166
<b>10</b>	<b>Case Study Implementation and Evaluation</b>	<b>169</b>
10.1	Realizing Handover in Motion . . . . .	169
10.2	Realizing Gesture Control of the Quadcopter . . . . .	175
10.3	Realization of Requirements . . . . .	178
<b>11</b>	<b>Conclusion</b>	<b>181</b>
	<b>Bibliography</b>	<b>185</b>



<b>List of Figures</b>	<b>195</b>
<b>Index</b>	<b>199</b>



## INTRODUCTION

Nowadays, industrial robots are an important factor increasing the productivity in many sectors. Especially in automotive, electrical and electronics industry, but also in further fields robot-based automation has continuously grown, with 229,261 industrial robots sold in 2014 (an increase by 29% compared to 2013) and a total stock of 1.5 million operational industrial robots at the end of 2014 (as recorded in the World Robotics 2015 report published by the International Federation of Robotics, [45]). However, most industrial robots are programmed in specialized, proprietary programming languages provided by the robot vendors that are mainly procedural and have a limited feature set. They provide the precision and reliability required for industrial robots, but hardly allow to benefit from the advances in software engineering achieved since the design of these languages.

In recent work by Angerer [2] and Vistein [103], an approach has been developed that supports programming industrial robots in a modern object-oriented standard programming language while maintaining precision and reliability properties of classical industrial robots. There, the main goals were to improve and speed up the development of robot applications by allowing to use state-of-the-art software tools and frameworks, along with an object-oriented modeling of the environment. This way, tasks and work pieces are clearly defined and can be exchanged, improving reusability of robot applications. Hoffmann [39] extended this work to service-oriented automation cells, introducing a service model on different levels to make entire robot-based automation systems more flexible. This is especially important for smaller lot sizes, when it is no longer acceptable to spend months of programming when a new kind of work piece should be processed, and for a *Smart Factory* [52] producing individualized products at lot size one.

In the context of *Industry 4.0* [52], a term introduced in the high-tech strategy of the German government as the fourth industrial revolution and combining *Smart Factory*, *Internet of*

*Things* and *Internet of Services* to create a virtual copy of the physical world [37], this software model and flexible programming approach is an important building block. However, for more flexible production, stationary industrial robots are not sufficient, but mobility and flexible logistics play an important role. Only if the exact flow of work pieces through the plant is no longer statically given, the full potential of a smart factory can be unlocked. To this end, “2,644 logistic systems were installed in 2014” [45], while “service robot suppliers estimate that about 16,000 mobile platforms as customizable multi-purpose platforms use will be sold in the period 2015-2018.” [45].

While mobile robots have been around for a long time, yet back in the 1970s with Shakey from the Stanford Research Institute [84], they are not yet widely used in manufacturing and pose new challenges that do not exist for stationary industrial arms. Obviously, mobile robots move around, and thus cannot be statically linked to their environment. Without cable connections and using only unreliable wireless networks, they cannot fully be controlled centrally, but need a certain amount of autonomy and on-board processing. Using multiple mobile robots thus brings up the topic of (software) distribution, especially when cooperation between the different manipulators is desired. Furthermore, mobile platforms typically have a driving accuracy that is by orders of magnitude worse than for industrial manipulators (that have a typical repeatability of less than 0.1 mm). Thus, absolute precision cannot be provided by the locomotion system alone, but requires sensors to measure the inaccuracy and handle it correspondingly.

Still, the further devices and objects expected in the environment and especially the work pieces that have to be processed are known or at least well defined in their blueprints, so object-oriented programming with domain modeling and modern software engineering approaches promise advantages for development. Thus, the goal and research question of this thesis is:

How can mobile robots be modeled and coordinated using an object-oriented software development process, coping with the challenges of uncertainty and distribution, and facilitating the cooperation of mobile manipulators?

Looking at the work of Angerer [2], Hoffmann [39], and Vistein [103], a powerful and extensible approach is available, that is however limited to stationary robots and does not handle the new challenges of mobile robots appropriately. Still, this approach supports important design goals of this thesis by already supporting robot arms as an integral part of mobile manipulators, and providing the advantages achieved through object-oriented programming.

Thus, it has been used as a basis for this work, however applying essential changes to consistently integrate the new aspects of mobile manipulators. Still, the main programming paradigm is left unchanged, allowing to write programs similar to the ones for industrial robot arms, but also to access new functionalities that are especially helpful for mobile robots.

---

The main contributions of this work are:

**Modeling different views of the world:** Working with multiple, independent mobile robots, each has a different knowledge about its environment. These knowledge differences are based on the sensors available to the individual robots. The proposed approach allows to model these different views, and facilitates consistent coordination of the robots, each working with its available data.

**Modeling sensor data and their geometric meaning:** The low precision of mobile robot locomotion and their use in not fully structured environments introduce geometric uncertainty. To handle this, sensors are added and the relationship between their measurements and geometric aspects is defined. Based on these definitions and incoming sensor data, the framework provides an automatic derivation of position estimates.

**Modeling expected outcomes and reactive behavior:** The more dynamic environment of a mobile robot leads to different possible outcomes of task execution. These possible outcomes are explicitly modeled and can be used in the definition of reactive behavior. The following steps reacting to the different outcomes can be defined and prepared incrementally or up front.

**Defining real-time reactions for coordinated tasks:** The approach allows to define real-time reactions to sensor data, and to synchronize the different actuators and sensors present in a mobile robot. This for example allows sensor-aware combinations of motions for the platform and arm of a mobile manipulator, but also guaranteed reactions to certain events.

**Supporting mobile manipulators:** For mobile manipulators, the approach facilitates an easy way of programming, providing means to handle kinematic restrictions of individual parts as well as redundancy of the complete mobile manipulator. Additionally, it allows to use advanced capabilities such as impedance control to mitigate position uncertainty for manipulation.

These contributions are achieved in a consistent object-oriented programming approach for the modeling and cooperation of mobile robots, respecting the special challenges of uncertainty and distribution. Describing this approach, this thesis has a clear structure. Starting with the basics of object-oriented robot programming and mobile robots and a description of the example applications used as a case study, the stage is set for the different interdependent facets required for the object-oriented modeling of a (mobile) robot application. With a focus on *task modeling*, different aspects of geometry, motions, real-time tasks and device capabilities have to be described. Shifting towards *mobile robotics*, motion execution in the presence of kinematic restrictions becomes important, as well as the additional aspect of sensor processing

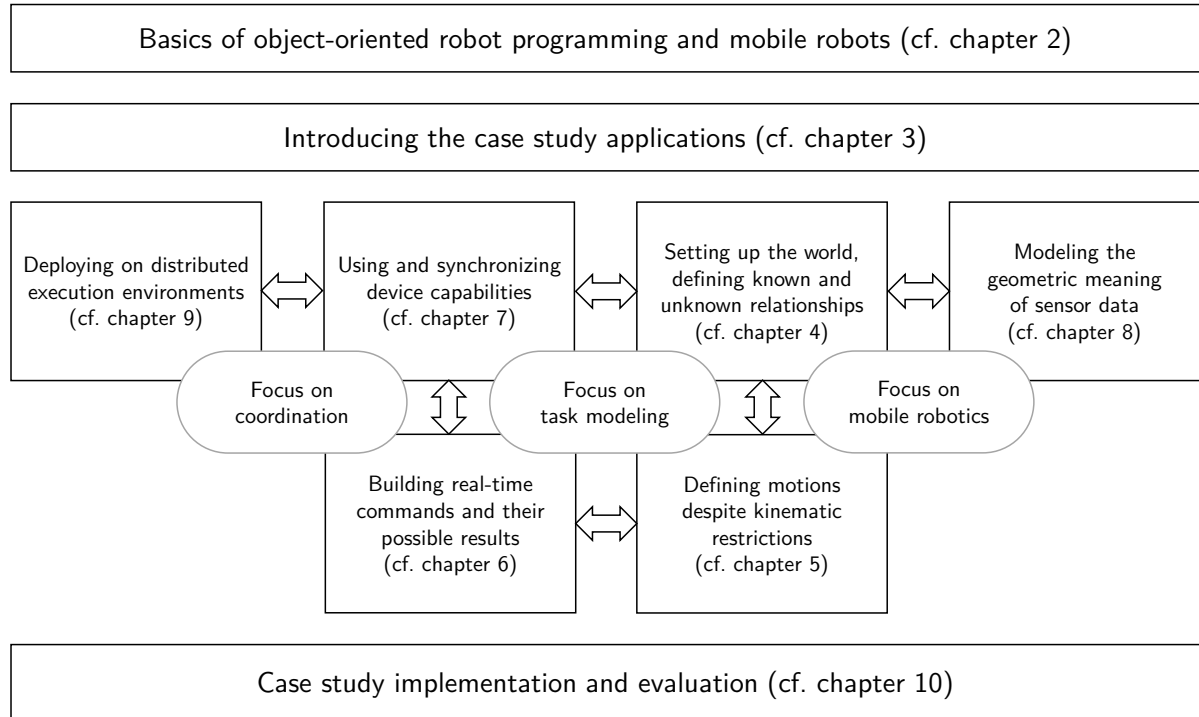


Figure 1.1: Structure of this work

based on and extending the modeled geometry. For *coordination* of mobile manipulators, the possible results of real-time tasks can be used for reactive behavior, while the synchronization of multiple devices gets special attention in the focus of distributed execution environments. Based on these modeling tools, the case study applications can be implemented and evaluated. Figure 1.1 gives an overview over this structure, along with references to the corresponding chapters. In detail, this work is structured as follows:

In chapter 2, an overview of the existing object-oriented approach for programming industrial robots is given. Additionally, the new challenges posed by mobile robotics are introduced – mainly distribution and uncertainty – and formulated as requirements to a software architecture. For mobile robots, the topics of software structure and distribution play an important role. These topics independently occur on three levels – real-time, system and application – and can be handled there independently for a mobile robot solution.

As an application example and case study, the programming and cooperation of KUKA youBots – small mobile platforms with an attached robot arm – and quadcopters is analyzed. Chapter 3 describes the used hardware devices, as well as the example applications of passing an object from one youBot to another while in motion and controlling a quadcopter through hand gestures.

As a basis for these applications, the environment set up has to be defined. In chapter 4, the

---

mechanisms for describing positions, motions and objects in the environment, modeling the structure of robots and defining the objects and devices used in an application are described. Additionally, this chapter (as well as the further chapters) describes the differences to the previous approach for programming industrial robots with respect to mobile robots, as well as a comparison to other popular robot frameworks and solutions.

Within the defined environment, it becomes possible to describe motions of robots as well as tool actions. Chapter 5 introduces the different types of motion possible, as well as further parameters beyond the specification of a Cartesian path that affect motion execution. These motions are common to industrial robots arms, mobile robots and quadcopters, while the respective device-specific limitations can be handled through the introduced concept of frame projectors.

These motion and tool action definitions together with an actuator that can execute them form the basis of executable command specifications. Chapter 6 describes these command specifications and their further properties, and explains the way how the specifications are transformed into a form that can be executed on real hardware. Additionally, these specifications can be combined, thus allowing to model larger tasks that can be executed with timing guarantees and without the risk of unintended delays caused by high CPU load on the computer running the application.

Chapter 7 shifts the focus from this low-level mechanism of command specifications to a more application and device centric view, modeling device capabilities and their synchronization. There, means of accessing device capabilities from applications are introduced, and ways to combine them according to common composition patterns to define more complex behavior. A special focus is put on reactive behavior, where the robot decides on its next steps based on sensor data or other stimuli that only become available while the task is already running, an aspect important for mobile robots that has not been handled thoroughly in the previous work.

The next important property of mobile robots – working in a less structured environment with uncertainty – is handled in the following chapter. While for industrial robots, the positions of work pieces and tasks are exactly defined and ensured through fixtures, mobile robots often work in an environment where this strict structure is not present. The main idea here is to use sensors – either integrated into the robot, or mounted in the environment – to resolve the uncertainty based on measurements, and to consistently update the world model – the representation of the application’s beliefs about its environment – accordingly. Chapter 8 describes how raw sensor data can be accessed using the proposed approach, and how to define the relationship between sensor data and aspects of the environment geometry. Additionally, estimators are introduced that handle sensor data to update the world model correspondingly.

Concluding the main aspects required to define an application, chapter 9 goes into detail about the deployment and execution aspects. Here, the software objects used to describe devices in application workflows are linked to real hardware devices that are connected to a single

or different computers, filling the last prerequisite to execution and handling the distribution part of cooperating mobile robots. Furthermore, this chapter describes the available execution environments that can be used for hardware devices or as simulation environments when the application is to be tested without real hardware.

Returning to the application examples from chapter 3, chapter 10 describes the implementation of the youBot interaction and gesture control applications according to the concepts introduced in the previous chapters, and revisits the requirements to the software given in chapter 2. Concluding this work, chapter 11 summarizes the accomplishments and gives an outlook.

Throughout this work, the first major occurrence of keywords with its definition is written in **bold face**, while further references to the keyword are written in serif letters. These keywords are also included in the index for easy access. Further keywords that are not defined in this thesis (e.g. names or elements of figures) are written in *slanted* letters.



## OBJECT-ORIENTED ROBOT PROGRAMMING AND MOBILE ROBOTS

Today, various different ways exist to program robot systems. On the lowest level, methods from control engineering are used to design the feedback controllers operating the robot. Here, model-based approaches such as Matlab [18] and Simulink [8] are available, while others prefer to directly write the corresponding code in programming languages such as C. Working on this level however requires a great amount of knowledge about the system and its dynamics, and is thus less applicable to people who see the robot as a tool for realizing their workflow or manufacturing process.

For these users, robot vendors have implemented high-level programming languages to describe the sequence of robot actions to be performed. Some of these languages describe the geometry or workflow graphically: For example, Lego NXT-G [55] allows to program LEGO robots by composing graphical blocks for individual tasks, defining control structures and reactions to events. For industrial robots, KUKA SIM [104] can be used to graphically define relevant robot positions in a 3D visualization environment, and to program motion sequences using these positions. Furthermore, textual programming languages are available: For ABB robots, RAPID [1] is available as a simple procedural language, while KUKA offers the KUKA Robot Language (KRL [76]) to program their robots. Due to their restricted nature, these languages can provide the precision and timing guarantees requested by the customers. However, they lack many concepts of modern programming languages that would allow extensibility, reuse or efficient development, as described by Angerer [2].

Another way of programming robot applications popular in robotics research is to use component-based systems [12, 13]. The idea here is to structure the software into components that provide certain functionality through a defined interface. These components can be interchanged with other components providing the same interface and functionality, and be reused in other applications requesting the same functionality. The components communicate with each

other through messages specified in the interface. The connection between the components and thus the decision which components communicate with each other is defined in the deployment phase, allowing to switch one component without modifying the others. This scheme provides better reuse and extensibility and also allows to develop new robot capabilities that use existing components in a new context or in new combinations. However, it has some drawbacks about flexibility and precision: Typically, the messages exchanged between the components are plain data containers, which carry no behavior. Thus, the advantages of object-oriented programming, such as inheritance and encapsulation of functionality [87] cannot be used. For example, when a symbolic planner encapsulated into a component is provided with actions describing possible execution steps (along with their pre- and postconditions), these actions cannot contain the logic required to actually execute the step. This way, the planner is limited to a set of known action types, for which the required specification information has been defined in the component interface, and cannot easily be used with further action types. Additionally, real-time aspects are typically handled in one of two extreme ways: Either, the frameworks consider real-time as important (cf. OROCOS [14]) but require that the application code is also implemented in a real-time safe manner, or they totally ignore this aspect for the communication between components (cf. OPRoS [47], ROS [82]), which reduces the precision or guaranteed reactivity of the robot solutions.

Other software development approaches focused less on developing new capabilities, but rather on making existing basic capabilities more easily accessible. These approaches started to bring object-oriented programming to the robotics domain. Early work starting in the second half of the 1990s on *MRROC+* [107], *ZERO++* [79] and the *Robotic Platform* [65] was based on C++ and began seeing robotics as a library to use in a full-featured object-oriented language. Later, starting in 2007, the *SoftRobot* project [40] provided a multilayer software architecture that allows to program industrial robots in modern, object-oriented programming languages such as Java or C#, while still providing real-time guarantees where required. The work presented in this thesis is based on the latter software architecture, introduced in the dissertations of Angerer [2], Vistein [103], and Hoffmann [39], and extends it to mobile robots, handling further requirements that arise in this domain.

Section 2.1 gives an overview of this software architecture used to combine modern object-oriented programming languages with real-time requirements for industrial robots, along with its design goals and achievements. Proceeding to mobile robots however, the existing design and implementation of this software architecture shows severe limitations. Section 2.2 describes the new requirements to the software architecture that occur with cooperating mobile robots, motivating the approach presented in this thesis. Apart from handling sensor data and uncertainty, one main requirement involves deployment and distribution. Section 2.3 gives an overview about the different levels on which the software for mobile robots can be structured and distributed.

## 2.1 The Robotics API – Object-oriented Programming of Industrial Robots

Analyzing typical applications in industrial robotics and the features provided in the programming languages developed by robot vendors [40] showed that determinism and real-time guarantees are required in parts of robot applications. This is true in the low-level execution part of robot controllers, where real-time guarantees are a prerequisite to the precision and performance expected from and delivered by industrial robots. But also some parts of the workflow of robot programs can be time critical, e.g. during the time when a welding torch is enabled. In this situation, it is not acceptable that the robot stops due to background tasks running on the robot controller. Otherwise, the background activity could lead to destruction of work pieces. However, between these real-time critical phases, times exist where the exact timing is not crucial to the correct execution of the task.

This insight forms the basis of the software architecture developed in the *SoftRobot* project. In the given situation, it is possible to write application code in a programming language that cannot provide timing guarantees, as long as it is possible to specify self-contained tasks for which execution with real-time guarantees is possible. In the given example, the task consists of switching on the welding torch, moving the robot along the predefined path, and switching off the torch afterwards. Between such tasks, the robot has to be in a stable condition where delays that occur in the application do not cause damage [40].

The software architecture consists of two layers (cf. figure 2.1). The upper layer is used for application programming without real-time guarantees, while the lower layer is responsible for performing the real-time critical robot control and task execution.

In the application programming layer, access to the concepts of robotics is provided through the *Robotics API*, an application programming interface provided by a *Robotics Application Framework* implemented as a software library. This application framework allows to model the physical world including robots in an object-oriented fashion. These objects provide their capabilities and allow to specify the tasks to be executed. More details about this framework can be found in section 2.1.2.

The real-time robot control layer accepts task specifications and is responsible for real-time control of the connected devices. Therefore, it contains a *Robot Control Core* implemented for a real-time operating system with an interpreter for the tasks, as well as hardware drivers to communicate with attached devices through their vendor-specific interface. Additionally, it provides feedback about execution progress and sensor data to the application layer, which can be processed there without real-time guarantees. For details about the reference implementation see section 2.1.1.

Using this software architecture, it becomes possible to implement robot applications that can execute tasks with real-time guarantees where required, while not limiting the available lan-

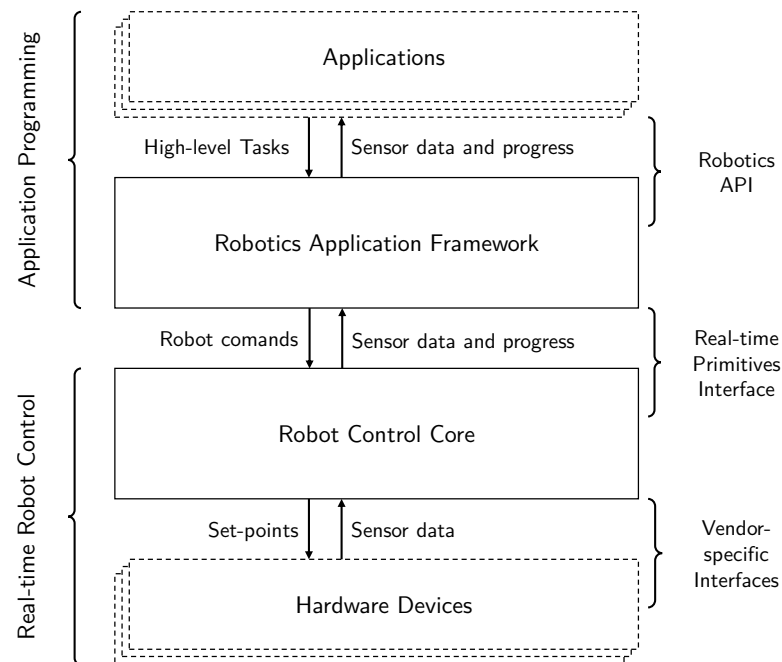


Figure 2.1: Two-layer software architecture separating application programming from real-time concerns (adapted from [2, 39, 103])

guage features and libraries. According to Angerer [2, chapter 11.6], this allows improvements in non-functional aspects of the development of robot applications such as usability, maintainability and testability, while a high level of performance and robustness can be maintained. In this context, modern development environments with their programming support become available, e.g. refactoring or debugging, as well as advances in language design (e.g. functional aspects integrated into object-oriented languages). Additionally, a wide variety of standard libraries can be used, e.g. for user interfaces, database access or communication with other systems. This way, robot applications can be integrated into business systems, or implemented using service-oriented architectures [39].

### 2.1.1 Specification and Execution of Real-Time Critical Tasks

In the reference implementation introduced by Vistein [103], the robot control layer is implemented as the so-called **Robot Control Core**. It is written in C++ and works on *Linux* with *Xenomai* extensions [35] to provide the expected real-time guarantees. Structurally, it consists of two major parts (cf. figure 2.2): A set of Device drivers communicating with hardware devices, and an execution engine for Data-flow graphs to perform tasks that are specified through a combination of Data-flow graphs and Synchronization rules.

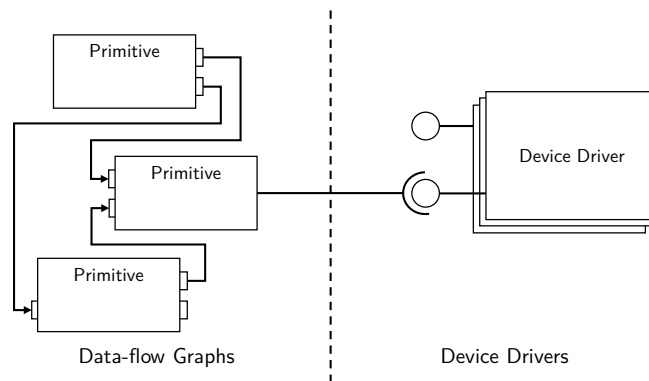


Figure 2.2: Structure of the Robot Control Core

### Interfacing the Hardware through Device Drivers

The **Device drivers** are responsible for interfacing the hardware devices attached to the controlling computer. They are implemented as components that can be loaded and configured at run time and provide interfaces for controlling the device and accessing its sensor data. Device drivers can either provide control at the level available directly by the hardware or firmware (e.g. velocity control of a motor), or can contain closed-loop controllers, if they want to provide higher level interfaces (e.g. position control for a motor that internally supports velocity control and position feedback). Device drivers can depend on each other, e.g. a driver for a robot arm connected through an EtherCAT bus can access an EtherCAT driver to communicate with the hardware device. The Robot Control Core provides a list of Device drivers with their interfaces to an application, and allows the application to add or remove Device drivers as appropriate for the task.

### Specifying Tasks as Data-flow Graphs

As part of the execution engine, Data-flow graphs (also called primitive nets or RPI nets) can be evaluated. The **Data-flow graphs** consist of calculation modules called Primitives and links. **Primitives** have an external interface consisting of input ports, output ports and configuration parameters, and an implementation that when evaluated reads the data from input ports and parameters, performs the corresponding calculation and writes the results to the output ports. Some Primitives reference Device drivers and can thus provide sensor data or handle set-points for the corresponding actuator. Configuration parameters are set initially and remain constant, while the values of the input ports can change at each invocation. Between an output port of one Primitive and the input port of another Primitive, a link can be established if the port types match. This link declares that the output value of the first primitive is to be used as an input value of the second primitive. This way, complex calculations that consist of multiple steps can be defined. Additionally, multiple primitives can be combined into a **Fragment**, which again has

input ports and output ports. Its implementation consists of a Data-flow graph, together with links from the Fragment input ports to input ports of the inner Data-flow graph, and links from outputs of the Data-flow graph to the outputs of the Fragment. Additionally, Fragments have an activation input that determines if the Fragment should be evaluated.

Looking at the ports, various data types are available: For logic decisions, *Boolean* values are appropriate, while for calculations *Integer* for integer arithmetic and *Double* for floating point are used. Additionally, composed types exist consisting of simple or further composed types. Examples include *Vector* for a vector in three-dimensional Cartesian space, or *Frame* to describe a translation and rotation in Cartesian space. Furthermore, fixed-size arrays are available as data types. *DoubleArray* can thus be used to transfer a fixed number of *Double* values.<sup>1</sup>

Generally, all data types support a designated *null* value indicating that the value is not available. This value is used for the output ports of a disabled Fragment, but also for history Primitives that provide access to the value of a port some time ago, if the primitive was not active at this time. The other calculation primitives handle the *null* values using strict semantics (returning *null* if any input was *null*) or following the three-valued logic if applicable (e.g. *true*  $\vee$  *null* is still *true*, cf. [81]).

Active Data-flow graphs are executed in a cyclic manner, with a typical frequency of 500 Hz. To allow this, all computation modules are required to have guaranteed timing bounds for their maximum execution time, to make sure that evaluation of the Data-flow graph will always complete in less than 2 ms. To execute the Data-flow graph, all primitives are evaluated sequentially, propagating the result values along the links. Therefore, a topological order of the primitives is established beforehand, so that all primitives that appear in links to the inputs of a primitive will be executed before that primitive. This sorting is possible because the Data-flow graphs are generally required to be acyclic. An exception are *Pre* Primitives, which are available for each data type (e.g. *BooleanPre*, *VectorPre*) and allow to transfer the value of an output port to an input port, but delayed to the next evaluation of the Data-flow graph. If at least one *Pre* primitive is contained in a cycle, the topological sorting can still occur, because the *Pre* primitive introduces a natural point for splitting the cycle. The actual execution is performed in three steps: First, all primitives that represent sensors are requested to read the required sensor data from their referenced Device drivers. Additionally, in this phase the *Pre* primitives transfer the value read in the last execution cycle to their output port. Next, all primitives are executed in the calculated order, transferring the computed values along the links. In the case of Fragments, the contained Primitives are executed in a topological order of their own, providing the output values for the fragment. Finally, all actuator primitives are requested to write their results to the referenced Device drivers. This way, a complete execution step has been performed, processing the sensors by applying calculations that yield set-points to command the devices.

---

<sup>1</sup>The fixed, predefined size is required due to the real-time requirements, as dynamic memory allocation cannot be performed with timing guarantees.

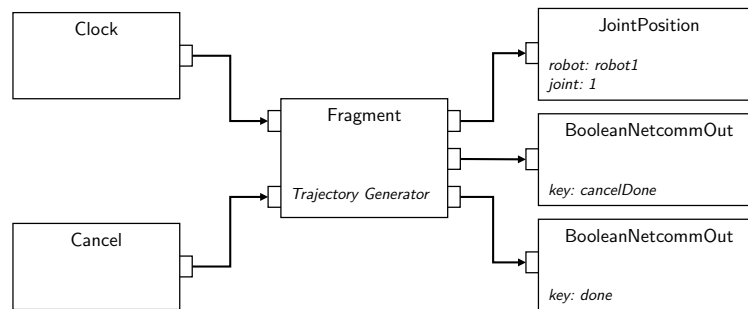


Figure 2.3: Example data-flow graph for a robot joint motion

For working with the data types and device drivers, different groups of computation primitives are available: A lot of primitives are provided to inject constant values of different data types and to perform basic computations on these. For example, *DoubleValue* allows to introduce a constant value into a data-flow graph, while *BooleanAnd* links two *Boolean* values through a logical *and* operation and *DoubleArrayGet* accesses a certain value within an array of *Double* values. Other primitives are stateful and calculate their result based on inputs and internal state, such as *Clock* that works as an integrator and returns the run-time of the data-flow graph in seconds. To allow communication with partners outside the data-flow graph, *Netcomm* primitives are used to exchange data with other data-flow graphs or robot applications. A *BooleanNetcommOut* primitive provides one *Boolean* value to the environment, while *DoubleNetcommIn* accepts a *Double* value from outside and *VectorInterNetcommIn* reads a *Vector* provided by another data-flow graph through its corresponding *VectorNetcommOut*. To interface devices, sensor and actuator primitives are used, such as *JointMonitor* to read the position and velocity of a robot joint or *JointPosition* to command a new position set-point.

Using these Primitives, it becomes possible to describe control laws, motions and tasks, taking into account sensor data and time progress to calculate device set-points. Figure 2.3 shows an example data-flow graph for the motion of a robot joint. It contains a *Clock* primitive tracking the time progress and a *Cancel* primitive to notify the data-flow graph if the motion is to be canceled. Additionally, it includes a *Fragment* that uses the time progress and optional cancel request to compute a position set-point for the robot joint using the available calculation primitives, thus implementing a trajectory generator. The computed set-point is passed on to the *JointPosition* primitive that forwards it to the corresponding device driver, thereby commanding the device (in an open loop fashion). Additionally, the trajectory generator offers one output port to inform that the motion has completed, and one output port reporting that the motion has ended due to a cancel request. These output ports are connected to *BooleanNetcommOut* primitives that make the value available under the given key (*cancelDone* and *done*) for further processing outside the data-flow graph.

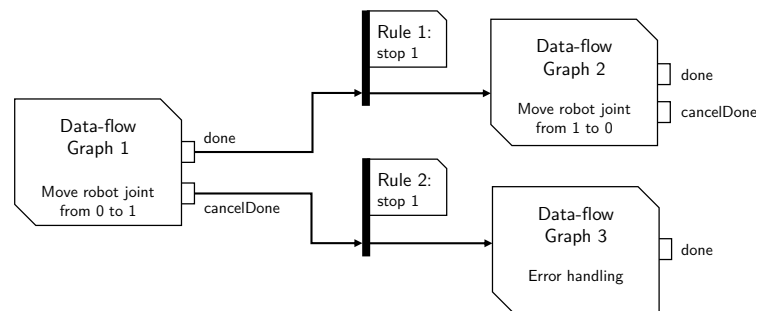


Figure 2.4: Example synchronization rules for motion sequences

### Controlling Task Execution through Synchronization Rules

To specify when a Data-flow graph is executed, **Synchronization rules** are used. A Synchronization rule consists of four parts:

**Synchronization Condition** A Boolean expression determining when the Synchronization rule should become active. The expression can use *and*, *or* and *not* to combine values provided by Data-flow graphs through *BooleanNetcommOut* primitives, according to the rules of three-valued logic. If the expression evaluates to *true*, the Synchronization rule is activated.

**Stop Nets** A (maybe empty) set of Data-flow graphs that are to be aborted once the condition is met. These Data-flow graphs are not informed, but will no longer be executed in the next execution cycle.

**Cancel Nets** A (maybe empty) set of Data-flow graphs that are to be canceled. Once a Data-flow graph is canceled, all *Cancel* primitives it contains return *true*, telling the Data-flow graph that it should bring the controlled devices into a stable state and come to an end.

**Start Nets** A (maybe empty) set of the Data-flow graphs that are to be started. If the stopped Data-flow graphs had control over any devices, these devices have to be handled by the newly started Data-flow graphs, to avoid that the devices are left in an unsafe state.

For data-flow graphs such as the one shown in figure 2.3, Synchronization rules can be used to specify the execution order, as shown in figure 2.4. Here, an outside view of the Data-flow graphs is given, showing them along with their possible results provided through the *BooleanNetcommOut* primitives. In addition to *Data-flow Graph 1* representing the motion from the previous example, one further motion (*Data-flow Graph 2*) and an error handler (*Data-flow Graph 3*) are used and linked through Synchronization rules. The Synchronization rules are shown as flags with arrows. The rule is activated once all results given by the inbound arrows are active. Then, it stops (or cancels, respectively) the Data-flow graphs given in the flag, and starts the Data-flow graphs given through the outbound arrows. *Rule 1* specifies that once *Data-flow Graph 1* reaches its *done* result, the motion (*Data-flow graph 1*) should be stopped and the



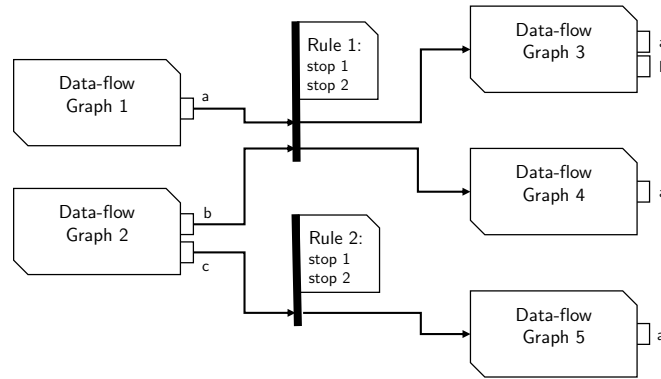


Figure 2.5: Example synchronization rules for device concurrency

next motion (*Data-flow Graph 2*) should be started. If however the first motion is canceled, *cancelDone* becomes true once the motion has stopped, and *Rule 2* replaces the motion with the error handler specified in *Data-flow Graph 3*.

In addition, Synchronization rules can be specified that affect multiple Data-flow graphs: In figure 2.5, *Rule 1* specifies that once *Data-flow Graph 1* reaches its result *a* and *Data-flow Graph 2* reaches *b*, both Data-flow graphs should be stopped, and *Data-flow Graph 3* and *4* should be started. *Rule 2* specifies to abort graph 1 and 2 if *c* is reached in *Data-flow Graph 2*, continuing with *Data-flow Graph 5*. These mechanisms can be used to synchronize motions of multiple actuators, guaranteeing that the corresponding Data-flow graphs are started and ended exactly at the same time (cf. sections 6.2.3 and 7.2.2). In the formalism introduced by Vistein [103], these rules are written as the following quadruples:

Rule 1:  $(G1.a \wedge G2.b, \{G1, G2\}, \emptyset, \{G3, G4\})$

Rule 2:  $(G2.c, \{G1, G2\}, \emptyset, \{G5\})$

Generally, the Data-flow graphs are used for low-level control and basic tasks, while the Synchronization rules are used for coordination by deciding when to switch to another behavior. Both Data-flow graphs and Synchronization rules are specified incrementally from an application without real-time guarantees, however with the guarantee that once Data-flow graphs and Synchronization rules have been accepted, they will be evaluated and executed with real-time guarantees. During the run-time of an application, new Data-flow graphs and Synchronization rules are issued once the application logic decides to do so. The Data-flow graphs typically specify one or multiple situations when the task is completed and the Data-flow graph should be stopped, as well as situations for a possible transition to the following Data-flow graph.

Additionally, bigger real-time critical tasks that consist of multiple Data-flow graphs can be composed: First, the corresponding Data-flow graphs are loaded and connected through the internal Synchronization rules. Afterwards, the task is started (or appended to the previous task) by adding the first Synchronization rule. This way, the complete set of Data-flow graphs and Synchronization rules is present once the first Data-flow graph is started, and thus the complete

task can be executed with real-time guarantees.

### 2.1.2 Object-oriented Application Development for Industrial Robots

While powerful, Data-flow graphs and Synchronization rules are tedious to write manually. However, they can be created automatically from task definitions. This is possible through the *Robotics API* which allows to specify tasks in a modern object-oriented programming language (Java or C#) and includes the required logic to create the Data-flow graphs. The *Robotics API* introduced by Angerer [2] includes six models used to specify different parts of the static and dynamic parts of a robot application. The following sections give a short overview over these models, while more details of their application to mobile robots follow in the following chapters.

#### Modeling Actuators and Sensors with the Device Model

The first model is an extensible *Device model* allowing to flexibly describe physical devices that are used within an application. These *Devices* can represent *Actuators* or *Sensors*, and provide access to the functionalities supported by the devices. This access is established through runtime adapters that link the *Device* to an execution environment such as an instance of the Robot Control Core.

From a software design point of view, the *Devices* make use of object-oriented features such as inheritance to model the hierarchy of devices, and composition for composed devices. The runtime adapter serves as a proxy used to abstract from the exact low-level access used to reference the device on the execution environment.

By modeling devices as software objects, much knowledge such as geometry and capabilities, but also the relationship between different devices is made explicit and can be used for application programming.

#### Specifying Real-time Transactions with the Command Model

Functionalities to be executed by actuators are expressed in the *Command model* describing real-time critical tasks. Instead of directly adding the capabilities of actuators to the corresponding software objects as methods, a schema following the command pattern [31] is used.

Executable tasks are expressed as a *Command*, consisting of an *Action* and an *Actuator*, each represented as an object. There, the *Action* describes *what* job to execute, while the *Actuator* gives the information *who* should execute the job. Using this pattern, new *Actions* can be introduced without modifying existing classes, thus allowing extensibility. Multiple *Commands* can be composed to build up larger real-time critical tasks. Finally, *Commands* can be executed, triggering the automatic transformation into Data-flow graphs and evaluation on the Robot Control Core [89].

### Handling Real-time Data with the State and Sensor Model

To incorporate more dynamic information into tasks, the *State model* and *Sensor model* are available. *States* represent discrete system conditions within a *Command*, which can be used to define system reactions. Additionally, *States* can be combined to describe more complex situations, and forwarded to an application to cause high-level behavior changes.

*Sensors* represent values provided by *Devices* or otherwise available in real-time. They can be used to create *States* defining system conditions, but also as an input to *Actions* to specify sensor guided motions. Similar to *States*, *Sensor* values can be combined in calculations evaluated in real-time, or forwarded to an application.

### Describing the Application Geometry with the World Model

To describe the geometric set-up of the environment, the *World model* is used. It defines notable positions in space, called *Frames*, along with *Relations* that describe the relationship and transformation between the *Frames*. These *Frames* are used to further describe *Devices* and their geometric relationships, but also as goal positions for robot tasks.

### Defining Physical Objects in the Entity Model

Hoffmann [39] extended the *World model* by an *Entity model*, further describing the physical objects and their composition. A physical object can therefore consist of other physical objects, but also contain *Frames* and *Relations* from the world model, and provide additional information, e.g. about shape and physical properties of the object. The *Entity model* thereby allows reasoning about the relationship between different physical objects, and also forms the foundation of a 3D visualization of the scene.

### Using Advanced Programming Features with the Activity Model

To simplify application programming and provide easy access to functions often used in industrial robotics, the *Activity model* is available on top of the *Command model*. It augments *Commands* with metadata about their resulting state, which can be used to plan successive tasks. This way, concepts such as motion blending or the transition between two tasks while a robot is in physical contact with the environment and exerting a force become possible and can be used in applications in an intuitive way. In the *Activity model*, composition is again possible to define greater real-time critical tasks, providing typical composition mechanisms such as parallel or sequential execution or sub-tasks triggered by given conditions of a main task.

## 2.2 Using Object-Oriented Programming for Mobile Robots

The aspects and solutions described in sections 2.1.1 and 2.1.2 allow to implement robot applications in modern object-oriented programming languages, while still providing real-time guarantees where required for precision or robustness. The use of data-flow graphs and synchronization rules proved to be an extensible and flexible mechanism to specify real-time transactions (cf. [103]). Furthermore, the separation of application logic and real-time aspects improves the usability of the programming approach, while still providing an extensible and reliable software framework (cf. [2]). The used object-oriented models allow to easily program robots by talking about concepts from the robotics and application domain (cf. [39]), instead of a purely robot centric view. Therefore, the approach fulfills the functional and non-functional requirements described by Angerer [2]. While these functional requirements originally targeted industrial robots, they remain important for mobile robots, and especially for mobile manipulators that include a robot arm. Thus, they have been condensed into the following Requirements 1 to 5 that will be revisited in the following chapters while adding a special focus on mobile robots.

**Requirement 1.** *Allow to specify the devices and manipulated objects in Cartesian space, along with further relevant positions (cf. FR-3 in [2]).*

**Requirement 2.** *Support an extensible set of robot motions in Cartesian and Joint space with configurable motion profiles (cf. FR-1, FR-2 and FR-4 in [2]).*

**Requirement 3.** *Support the exact execution of motions and arbitrary device actions, individually or precisely triggered by motion progress or sensor data (cf. FR-7, FR-8 and FR-12 in [2]).*

**Requirement 4.** *Allow to combine tasks of a single robot, of multiple robots and further devices, with or without stopping the robot in between (cf. FR-5 and FR-6 in [2]).*

**Requirement 5.** *Allow to access, process and convert sensor data in an application-defined way, and to use it for defining or influencing motions (cf. FR-9 to FR-11 and FR-13 to FR-14 in [2]).*

Additionally, the approach uses a standard, object-oriented programming language. This way, it becomes possible to integrate robot programs with other systems by using further standard libraries, but also to use the features provided by standard development environments and processes. This can be seen as the foundation of the reusability and productivity gains possible using the *Robotics API*, compared to classical languages for industrial robots (cf. [2]).

However, although the approach has successfully been used in examples for mobile robots [39], additional challenges are not yet handled in a reliable and consistent way. According to Chatila [16], perception, environment modeling (mapping), motion planning and control architectures are important topics that guided research on mobile robots and still influence current robot solutions. Looking at these topics, motion planning can be implemented on top

of a robot framework (cf. [39]) using the mechanisms and models it provides. However, to consistently allow perception and environment models, the framework has to support working with uncertain data for processing the sensor readings.

**Requirement 6.** *Describe and consistently integrate geometric data that is not initially and statically known, but is only measured through sensing.*

Mobile robots are typically used in environments that are not structured and unchanging, but rather complex and changing (cf. [70]). In easiest cases, this is caused by the inherent uncertainty about the robot pose. These environments require a control architecture that can integrate sensor data into planning, but also supports reactive behavior based on sensor readings.

**Requirement 7.** *Allow the specification of reactive behavior that allows fast reactions to environment changes and sensor data.*

Proceeding to multiple distributed mobile robots, further challenges appear (cf. [78]). With cooperative mobile robots, software integration, communication, and coordination of task execution become important aspects. Depending on the context and goal, controlling multiple robots can either be done from a single application in a strongly cooperative scheme, or from multiple independent applications for each robot. Therefore, the framework should support accessing multiple distributed robots, but also the cooperation of multiple applications.

**Requirement 8.** *Support distributed devices, some of which may be connected to different computers.*

These requirements are not yet handled in the existing work concerning the Robotics API, leading to problems when used with mobile robots or in situations where the environment is not precisely known. To handle these shortcomings, the existing concepts are adapted and extended to cope with situations where uncertainty or distribution become more relevant, to make the same advantages of the approach available to the programming of mobile robots. The following chapters describe these changes and extensions, along with their relation to approaches from other robot frameworks and literature.

## 2.3 Software Structure and Distribution for Mobile Robots

When looking at the software design for the cooperation of mobile robots, multiple levels can be identified where structuring and also distribution is independently possible (cf. figure 2.6). These different levels have previously been published based on an application example [88, 90] and are here described in a more abstract form. Additionally, existing component frameworks for robots are classified with respect to their focus and capabilities on these levels, and foundations of the design decisions for mobile robots in the proposed framework are introduced.

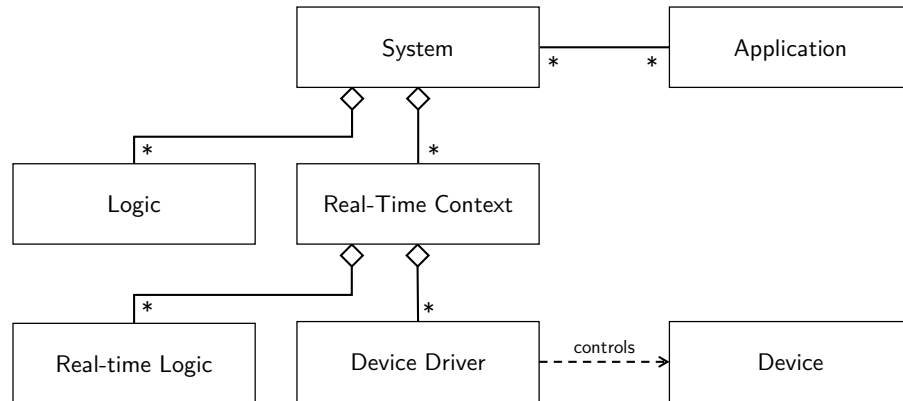


Figure 2.6: Software structure for distributed robots with real-time guarantees

Starting with the individual hardware devices used, each one requires a *Device Driver* to make it accessible in the software world. The *Device Driver* is defined as “the component that communicates with the hardware device through the vendor-specific interface”<sup>2</sup> [88]. It is responsible for data exchange with the hardware device, and to provide the device functionality to the surrounding software components.

The *Device Drivers* can be structured into *Real-time Contexts*. A *Real-time Context* in this sense is defined as “the components between which data transfer and coordination occurs with given timing guarantees” [88]. Depending on the implementation, one *Real-time Context* can span one *Device Driver* or multiple *Device Drivers*. Additionally, further computation components can be included in a real-time context. These can hold real-time logic to perform discrete task switching or continuous control, processing the data available from the device(s).

Multiple *Real-time Contexts* can be grouped into a *System*, together with further computation logic. A *System* includes “the components between which all knowledge is shared” [88]. Within a *System*, every component is allowed and able to access any other component’s data, and to send commands to each other. However, this data transfer does not provide the timing guarantees available in a *Real-time Context*.

To send commands to *Systems* and coordinate behavior, *Systems* can be used in *Applications*. An *Application* is defined as “the components that coordinate a work flow executed by the systems.” [88]. *Applications* can access one or multiple *Systems* and issue tasks that are to be executed by the *Systems*. The behavior of cooperating mobile robots is then defined by all active *Applications* and their coordination, which leads to the desired goal.

Depending on requirements and technical limitations, different software structures can be used on the *Real-time Context*, *System* and *Application* level. Looking at the deployment, the chosen structure can be distributed on each of the levels, so that a *Real-time Context*, *System* or

<sup>2</sup>Although the definitions talk about components, the ideas are not only applicable to component-based robot frameworks, but also to object-oriented robot programming, interpreting the different software parts as components.

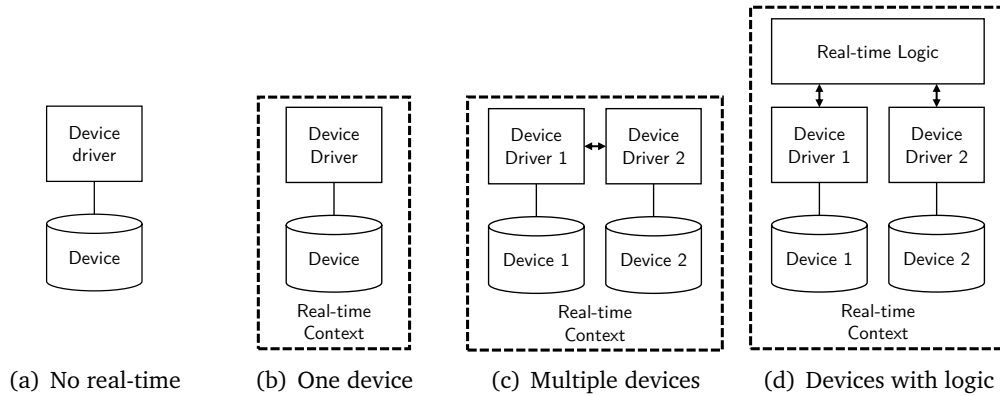


Figure 2.7: Examples of software structure on the real-time level

*Application* can be split to multiple computers.

The following sections describe the different structuring and distribution options on the real-time (cf. section 2.3.1), system (cf. section 2.3.2) and application (cf. section 2.3.3) level, along with the advantages and challenges that arise from these design choices.

### 2.3.1 Control and Deterministic Reactions – the Real-Time Level

When working with multiple devices, different ways of structuring exist. The smallest version is to implement device drivers without real-time in mind (cf. figure 2.7(a)). This way, simple control of the device can be achieved (if the vendor-specific interface provides an access mode without real-time requirements). It is then possible to issue a command to the device that will be executed, however without guarantees how long it will take before the command starts. For example, a set-point for a robot arm expressed in configuration space can be commanded, or (if supported by the device) a sequence of configurations specifying a more complex trajectory. However, guaranteed reaction to events (e.g. to avoid collisions) are only possible as far as they are already implemented within the device. In this case, the device driver does not belong to a real-time context, thus this case will not be regarded in the following classification.

To form a real-time context, real-time capable device drivers are required. To achieve this, the software environment has to provide a guaranteed amount of determinism. Therefore, real-time operating systems such as VxWorks [77], QNX [38] or Linux with RTAI [68] or Xenomai [35] extensions are typically used. Within these operating systems, certain tasks can run with real-time guarantees, i.e. with the guarantees that cyclic tasks will be executed at the given frequency and with bounded jitter, that the execution will take a deterministic execution time and will not be interrupted by lower-priority tasks.

Using a real-time operating system, a device driver can be implemented with real-time guarantees (cf. figure 2.7(b)). The device driver can then deterministically provide the device with new set-points. This allows to execute custom trajectories generated at run time, and

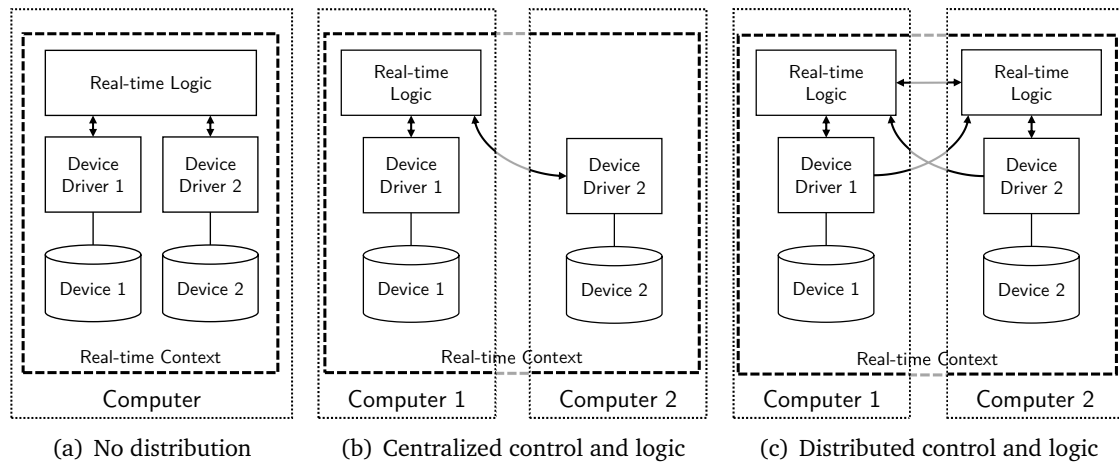


Figure 2.8: Examples of software distribution on the real-time level

also to incorporate feedback control that makes use of the sensor data available in the device. However, real-time guarantees are still limited to the one device, and exact synchronization between different devices or real-time reactions to other devices are not possible.

To exactly synchronize multiple devices or perform combined control, they have to be merged into a common real-time context (cf. figure 2.7(c)). Within this context, it becomes possible to use sensor data from one device to control a second device, or to execute combined motions of both devices. For example, a synchronized motion of a youBot arm and platform can be performed, using the platform to compensate for the kinematic limits of the arm.

However, for specifying such tasks, it becomes helpful to introduce further computation components in the real-time context, instead of implementing all required cooperation schemes into the device drivers (cf. [91]). These computation components then communicate with the device drivers (cf. figure 2.7(d)) and hold the real-time logic deciding when to switch between different tasks for reactive behavior, as well as implementations for the control laws describing the task-specific continuous dynamics of the controlled actuators.

Generally speaking, increasing the real-time context improves the capabilities of the corresponding devices, however with the drawback of higher complexity. This becomes clear when looking at the issue of distribution: While it is relatively easy to create a real-time context that spans all devices that are connected to one single computer using shared-memory communication (cf. figure 2.8(a)), adding distribution makes things more complex. To provide the requirements for a common real-time context spanning two computers, a reliable, deterministic and high-performance means of communication is required. Then two general options are possible: Either, all data is transferred to a central computer performing the required computations, which then sends the control set-points to the computers controlling the devices (cf. figure 2.8(b)). While easier, this method has the drawback that the central computer requires enough computation power to perform all calculations. The other way is to let all computers



do the calculations for their devices (cf. figure 2.8(c)). In this case, all the required sensor data has to be transferred between the computers, while each computer uses it to generate set-points for the local devices. Additionally, state or progress information has to be transmitted to synchronize the tasks executed on each of the computers. All this is possible using wired connections, e.g. a dedicated network or field bus protocols such as EtherCAT [48], but becomes problematic between mobile systems where only wireless connections are acceptable.

Looking at popular robot frameworks, *OROCOS* [14] focuses on control and on the real-time level. Typically, all components are composed into a single real-time context, using the real-time capable communication facilities provided by the *OROCOS* framework. Distribution of the real-time context is possible over the network, using the provided *CORBA* transport or another application-specific implementation. To combine multiple robots into a single real-time context, wired connections are typically used to guarantee deterministic communication, sometimes even for mobile robots (cf. [57]).

The SoftRobot approach [40] as well as the extended approach proposed in this thesis also provide real-time guarantees, and allow to specify tasks and real-time reactions between the different devices using a modern, object-oriented programming language. Both approaches use one real-time context for each computer that controls devices, so that the real-time context on each computer spans all devices physically connected to the computer, but is not distributed between multiple computers. For the given scenario however, this poses no limitation, because distribution is not needed within one KUKA youBot because it only contains one computer, and forming a real-time context spanning two youBots is not possible because the youBots lack means of deterministic wireless connection. Additionally, the reduced precision of mobile robots compared to industrial robot arms limits the advantages of using one real-time context, because cooperation purely based on time synchronization and common trajectories (as mentioned in section 7.1.2 and suggested by Angerer [2] and Vistein [103]) is not sufficient for the expected cooperation precision. Thus, this level of distribution is not in the focus of this work.

A different situation may arise for larger robots such as the *WillowGarage PR2* [20] or the *DLR Justin* [11] that contain more actuators and multiple on-board computers, where the increased work load or connectivity may require to distribute the work between multiple computers. There, the use of distributed real-time contexts with decentralized calculation can become helpful, which can be implemented based on the concepts presented in this thesis: Complex robot tasks are transformed into groups of independent Data-flow graphs (cf. section 6.2.2) along with separate Synchronization rules (cf. section 6.2.3), providing clear separation criteria for decentralized calculation as well as explicit data dependencies for switching between discrete behaviors. These form the basis for distributed real-time logic on multiple computers, keeping the data dependencies explicit and manageable.

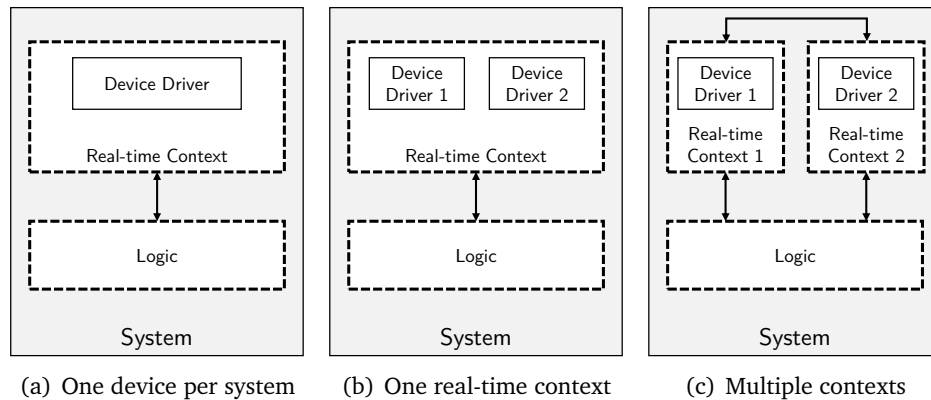


Figure 2.9: Examples of software structure on the system level

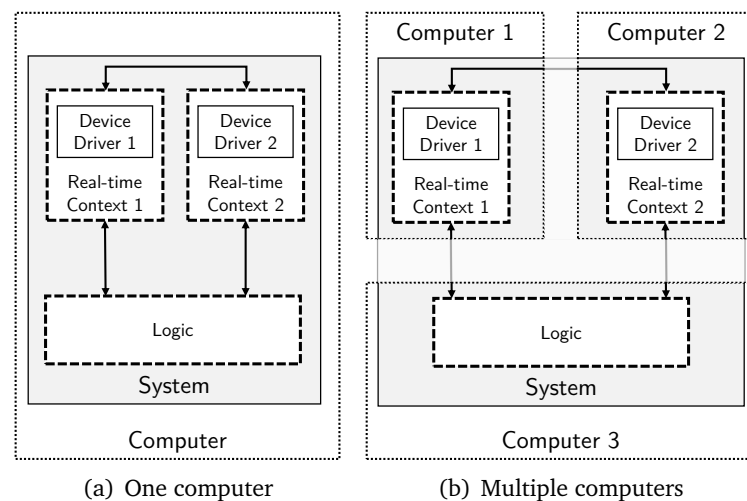


Figure 2.10: Examples of software distribution on the system level

### 2.3.2 Freely Communicating Components – the System Level

As a next method of structuring, real-time contexts can be combined into systems, making their data available to each other without real-time guarantees. The smallest option is to use one real-time context with one or multiple devices as a system (cf. figures 2.9(a) and 2.9(b)). Thus, the real-time context is limited to all the data available in real-time as well as data received from the application(s), but cannot access other devices that belong to other real-time contexts (or systems). The second option is to combine multiple real-time contexts into one system (cf. figure 2.9(c)). This way, all data from the other real-time contexts can be accessed, however in a best-effort way without real-time guarantees.

A system can be distributed (i.e. span multiple computers), as long as reliable (but not necessarily deterministic) means of communication exist to transfer the data from one computer to another (cf. figure 2.10). This is possible through wired as well as wireless networks, as long

as enough bandwidth is available to transfer all required data, and transmission outages or time delays are sufficiently small.

When programming applications for multiple devices, working with a single system simplifies work, because all information is assumed to be available everywhere. Thus, a single, consistent world model can be used including all devices and their environment. However, some disadvantages exist when using bigger systems: On the one hand, the amount of data present in the system, and with it the required amount of communication, increases. Then, network bandwidth can pose a limit to the system size. This can especially be a problem if many different robots are to cooperate with each other at different times – then all would have to be in one big system. Furthermore, programmers have to be aware that some parts happen with real-time guarantees while system communication introduces time-delays in other parts, which may lead to unreliable behavior of the cooperating robots. Finally, “political” reasons may be a reason for separate systems, if the devices belong to different parties: Some owners may not want all the data present in their robot system to be made available to all other robots, or may not want every other robot to be able to control their robot. In this context, issues such as trust or access control become important, which are not compatible with the concept of systems where everything is shared.

Looking at existing software frameworks, *ROS* [82] aims to allow transparent distribution on the system level, however without emphasis on real-time guarantees for the communication between nodes, so real-time contexts can only exist within single nodes. Many device driver nodes are implemented without focus on real-time, while other nodes do provide a real-time context for a single device (or even multiple devices, such as the youBot arm and platform). To achieve real-time guarantees within nodes, they can be implemented using *OROCOS*. All nodes that belong to the same *ROS master* can be seen as a system, because their communication between the different nodes is possible without limitations. On this level, *ROS* allows distribution over multiple processes and also over the network, because all messages passed by nodes can be serialized and transferred between different computers. This distribution is transparent to the user and node implementations, because *ROS* internally uses the configured *ROS master* to find the address of the communication partner.

In contrast, the proposed approach uses no mechanism for combining multiple real-time contexts into systems. The main reasons for this are the mentioned problems concerning scalability and reliability, when abstracting from network communication without real-time guarantees can lead to unpredictability. However, some data from one real-time context can explicitly be made available in another real-time context with appropriate modeling of unreliability and time delays.

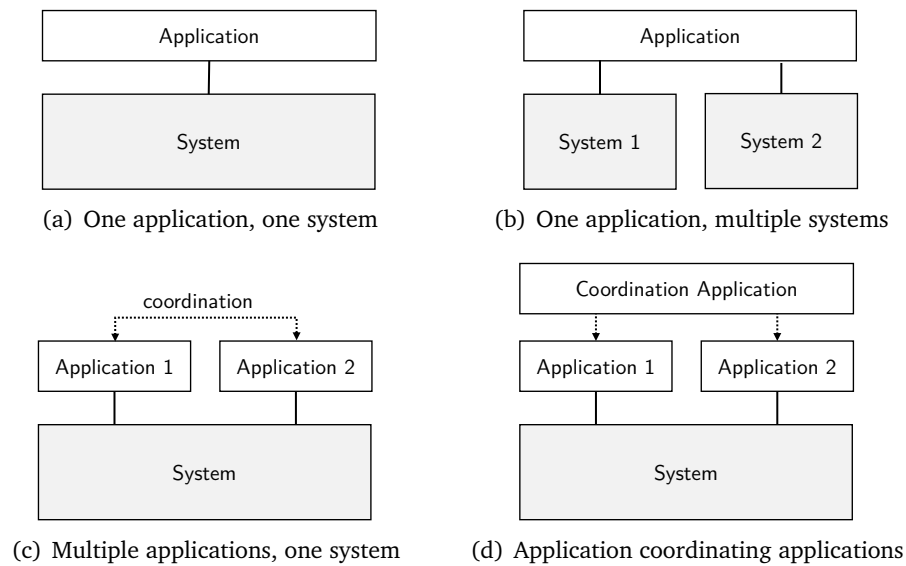


Figure 2.11: Examples of software structure on the application level

### 2.3.3 Coordinating What Happens – the Application Level

To perform the desired behavior, systems have to receive tasks from applications. Here, the relationship is less constrained (cf. figure 2.11): A single application can control multiple systems, but a system can also be controlled from multiple applications (as long as the controlled devices are always mutually exclusive).

The easiest situation is when a single application controls a single system (cf. figure 2.11(a)). Then, the entire workflow is described in one place, and the application can work using a consistent view of the world (as known in the system). If the system contains multiple real-time contexts, some limitations may still be in place that restrict the availability of real-time synchronization.

When multiple systems are used, the workflow can still be expressed in a single application (cf. figure 2.11(b)). Then however, the application has to be aware of the system boundaries, i.e. which information is known in which system. Some features may not be possible in this situation, e.g. to perform synchronized tasks on devices belonging to different systems, or to provide real-time guarantees for some coordination (because different systems imply different real-time contexts). An application written to work with multiple systems however can still work if the devices are within one system, as combining systems does not reduce the possible feature set. To make this possible, the decisions which systems should be accessed by an application and which devices belong to which system should not be part of the application logic, but rather be defined independently as a deployment aspect.

Another way is to describe the robot behavior through multiple applications (cf. figure 2.11(c)). Then, each application is responsible for coordinating the behavior of a subset

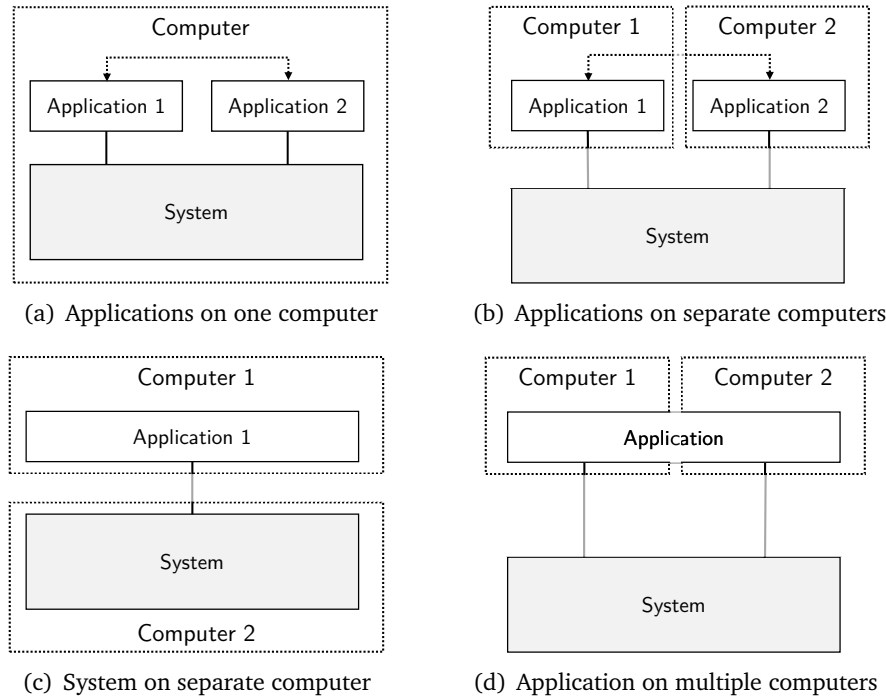


Figure 2.12: Examples of software distribution on the application level

of devices, while the interaction is governed by communication between the workflows. In this situation, the different applications can again address a single system or multiple systems, depending on the underlying structure. Finally, a single application can be used that coordinates multiple applications which are in turn responsible for giving low-level tasks to devices available in systems (cf. figure 2.11(d)). This situation however can be seen as similar to the single application case, when interpreting the coordinated applications as parts of the corresponding systems.

When looking at distribution, different options arise: Using more than one application, it is possible to assign the individual applications to multiple computers (or at least operating system processes, cf. figure 2.12(b)), or to keep them within one (cf. figure 2.12(a)). If the applications are distributed, coordination or data transfer has to be made explicit through network or inter-process communication. This complicates some matters, such as working on a common, consistent world model, but can be required if the different systems controlled are separated due to political reasons.

Another way of distribution is to split a single application onto multiple computers (cf. figure 2.12(d)). This can transparently be achieved through programming language mechanisms such as remote procedure calls or service-oriented architectures, and does not have significant influence on the design of the application.

As a third way of distribution, applications can be executed on a computer that does not

belong to the controlled system (cf. figure 2.12(c)). This is possible as long as the application communicates with the system through a network connection, or the computer running the application can easily be added to the controlled system.

Looking at *OROCOS* and *ROS*, neither of them natively supports a clear separation of applications from systems or real-time contexts. Using *OROCOS*, applications are typically implemented as real-time components or reside outside *OROCOS* and access provided devices through a *ROS* bridge. In *ROS*, applications are nodes that belong to (exactly) one system.

In the proposed framework, applications are implemented in Java and can use the distribution mechanisms provided by its runtime environment. Applications can work with multiple systems (each consisting of one real-time context), however this decision is made through a deployment configuration deciding which devices belong to which real-time context, and can be based on functional and non-functional requirements. The approach aims to make major parts of software development independent from this change, especially by defining the application geometry as well as robot tasks in a way that abstracts from the fact whether objects and devices are controlled from the same application or another. This way, applications are independent from their deployment, as long as they only use features that are possible in the given situation, and can thus be used with a single system even if programmed for multiple ones. This advantage will become obvious in the application examples, where the same application can be used with two independent youBot systems as well as with a simulation environment that handles both robots as one system.

## CASE STUDY: COOPERATING MOBILE ROBOTS

To illustrate the concepts introduced in this thesis and show the achievements and possibilities, two applications of mobile robots will serve as examples throughout this work. Both consist of a challenging but clearly specified task to be performed by the robots in a structured environment. These tasks are not novel in the sense that they show unprecedented robot capabilities – this work rather focuses on a software view how to clearly and consistently model the tasks in an object-oriented manner, along with the steps needed to enable the successful execution of these models in simulation as well as on real robot hardware.

In this thesis, mainly two types of mobile robots will be used, as described in section 3.1. The central robot will be the *KUKA youBot*, a mobile manipulator consisting of a mobile platform, a kinematically restricted robot arm and a two-finger gripper. More details about its hardware and existing software are explained in section 3.1.1. Additionally, a quadcopter will be used, carrying objects and thus extending the approach towards flying robots (cf. section 3.1.2).

To work in an environment that is not exactly known beforehand, sensing is employed, mainly using two types of sensors, as further described in section 3.2. Onboard the youBot, *Hokuyo* laser range finders are mounted that scan the distance to obstacles in a plane parallel to the ground, thereby allowing to detect objects to manipulate. Furthermore, a *Vicon* tracking system is used as an external optical tracking system that provides the position of the used actuators and of further objects equipped with markers with high precision.

Based on these actuators and sensors, two application examples are introduced in section 3.3. The first happens on the ground, where one youBot picks up an object placed on the ground and hands it over to a second youBot while in motion. The second example shifts the focus towards flying robots and includes gesture control of a quadcopter, along with cooperation between the quadcopter and a youBot.

These examples however are not the first work conducted on cooperative manipulation, or more specifically manipulation using youBots, cooperative work of youBots or the cooperation between youBots and quadcopters: Early work in 1996 by Khatib et al. [56] analyzed control and dynamics aspects of cooperative mobile manipulation, based on mobile manipulators available at that time. Based on youBots, control issues for mobile manipulation have been addressed by Keiser [54] by implementing torque control of a single youBot arm as an extension to the existing *ROS* implementation of the youBot driver, as well as an example of grasping an object while in motion. Dogar et al. [24] describe the control prerequisites to cooperatively carry a large object with multiple youBots, along with an implementation using *ROS*. Similarly, Tsiamis et al. [100] analyze the control aspects of cooperative manipulation using implicit communication, however limited to a simulation case in the *V-REP* simulation environment. With respect to planning, Knepper et al. [59] introduced *IkeaBot* as a solution implemented for *ROS* where a group of youBots cooperatively assemble a table, including a passage where the assembled table is flipped over by two youBots. Mueggler et al. [75] describe the planning aspects of a team consisting of a quadcopter and a youBot, where the ground robot is guided through an area with obstacles based on a map created by the aerial robot. In contrast, this work focuses on ways of software development and modeling for mobile robots, following the principles of object-oriented design, while still making sure that the resulting solutions can be used not only under perfect simulation conditions, but also in real-world environments.

### 3.1 Influencing the Environment – Actuators

As main participants in the case study, KUKA youBots and a quadcopter will interact. These two types of robot serve as actuators to influence the environment and to achieve the given goal.

#### 3.1.1 Mobile Manipulator – KUKA youBot

In the field of mobile robots, the KUKA youBot serves as a mobile manipulator (cf. figure 3.1). As such, it consists of a robot manipulator on top of a mobile platform, along with a two-finger gripper mounted as an end effector. This combination extends the range of capabilities well beyond their individual features, enabling manipulation in large work areas.

##### Mobile Platform

The youBot platform has a size of 58 cm by 37.6 cm and weighs 20 kg, providing a payload of 20 kg [61]. It contains a lead-acid battery with 24V, 5 Ah for an approximate run time of 90 min [61]. The platform is equipped with an omni-directional wheel system with four Mecanum wheels [44]. This allows the platform to move forward and backwards (if all wheels rotate in the same direction), but also to move to the side (if adjacent wheels rotate in opposite directions), to rotate in place (by moving the left and right wheels in opposite directions) and





Figure 3.1: The KUKA youBot mobile manipulator

to execute arbitrary combinations of these motions. This way, the platform can freely move and rotate in a plane and thus provides three degrees of freedom.

The wheels can be accessed through an EtherCAT bus [48], where the individual wheels can be configured for position, velocity or torque control and commanded with a cycle time of 1 ms. The platform houses an integrated controller PC with an Intel Atom Dual-Core CPU and 2 GB of RAM, supporting Linux and connected to the EtherCAT bus of the platform. To control the youBot, the internal PC can be used to communicate with the actuators and perform tasks. For low-level control, an object-oriented C++ API [62] exists, offering access to youBot control on the EtherCAT layer. For higher-level access, a ROS wrapper [85] for the C++ API is provided to integrate the youBot into the ROS ecosystem. As an alternative, the reference implementation developed along with this thesis can be used on the integrated PC to interface the youBot, offering options for real-time control and reactions, while allowing to write high-level applications in Java.

#### **Articulated Arm**

On top of the mobile platform, a robot arm with five degrees of freedom is mounted. It offers a payload of 0.5 kg within a work envelope of  $0.513 \text{ m}^3$  [61]. Due to its kinematic structure, it cannot reach each position with arbitrary orientation, but is limited to poses where (from a top view) the gripper points towards or away from the arm's center. The arm is also attached to the EtherCAT bus and can thus be controlled from the internal PC.

### Two-finger Gripper

At the end of the arm, a two-finger gripper is mounted. It offers a stroke of 2 cm and is shipped with parallel fingers. To better support grasping round objects, Festo FinGrippers [7] can be mounted. Their flexible part wraps around the grasped object, providing secure grip along with mechanical compliance to correct minor positioning errors.

### Mobile Manipulator

Combining platform, arm and gripper, the youBot can serve as a mobile manipulator. With the three degrees of freedom for the platform and five for the arm, the manipulator offers a total of eight degrees of freedom. This way, the youBot has no systematic limitations within its working envelope, but can achieve the full six degrees of freedom of Cartesian space. In addition, the youBot has two further degrees of freedom that allow it to change the orientation of the mobile platform as well as the distance between gripper and the platform's center, while keeping the position of the gripper in the world constant. This redundancy – and the ability to exploit it – is an important aspect of mobile manipulation, making mobile manipulators more flexible.

However, the precision and bandwidth of the arm and platform differ: While the arm offers a repeatability of 0.1 mm [61], the precision of the platform differs with the type of ground worked on, however ranges orders of magnitude worse. This way, using arm and platform increases the work space, but reduces the possible accuracy and motivates the need for sensing to compensate for these limitations.

For the case study, the youBot is equipped with two Hokuyo laser range finders (cf. section 3.2.1) for environment sensing and object detection that are connected to the internal PC. Additionally, an *Edimax EW-7811USC* adapter is mounted on the youBot to access a 5 GHz Wi-Fi network (established by a *Realtek RT-AC66U* router) used to transfer Vicon tracking data (cf. section 3.2.2) to the youBot, as well as to communicate with the application running on an off-board computer (that coordinates the different robots).

Furthermore, a radio receiver based on an *ATmega328* microcontroller and an *nRF24L01+* wireless transceiver is connected to the youBot, wired to trigger the soft-stop functionality of the youBot EtherCAT bus and notify the on-board PC when a stop command is issued from a wireless transmitter. This receiver serves as a safeguard for experiments with multiple cooperating youBots, making sure that all robots can be stopped when an unexpected situation occurs.

#### 3.1.2 Flying Robot – r0sewhite Saphira Quadcopter

As a flying robot, a Saphira quadcopter by r0sewhite [80] is used. It has a frame size of 24 cm and is equipped with 6" propellers. As a flight controller, the Autoquad M4 [5] board is used. The quadcopter is controlled through the *DSM2* protocol using a Deltang Tx1-K1 transmitter



Figure 3.2: The r0sewhite Saphira quadcopter

module [98], or for debugging using a Spektrum DX9 remote [25] through the trainer port and using the master override function.

The quadcopter is equipped with a 3D printed plastic carrier with foam rubber mounted on top (cf. figure 3.2). It allows to place objects such as glasses on top of the quadcopter, and makes sure they don't slip off during flight. Additionally, it is equipped with infrared reflective markers for tracking in the Vicon system.

## 3.2 Observing the Environment – Sensors

In addition to the actuators and their integrated sensors, further sensor systems are employed. These include Hokuyo URG laser scanners mounted on the youBot, as well as a Vicon tracking system set up in the room where the experiments were conducted.

### 3.2.1 Onboard Sensing – Hokuyo URG-04LX-UG01 Laser Scanner

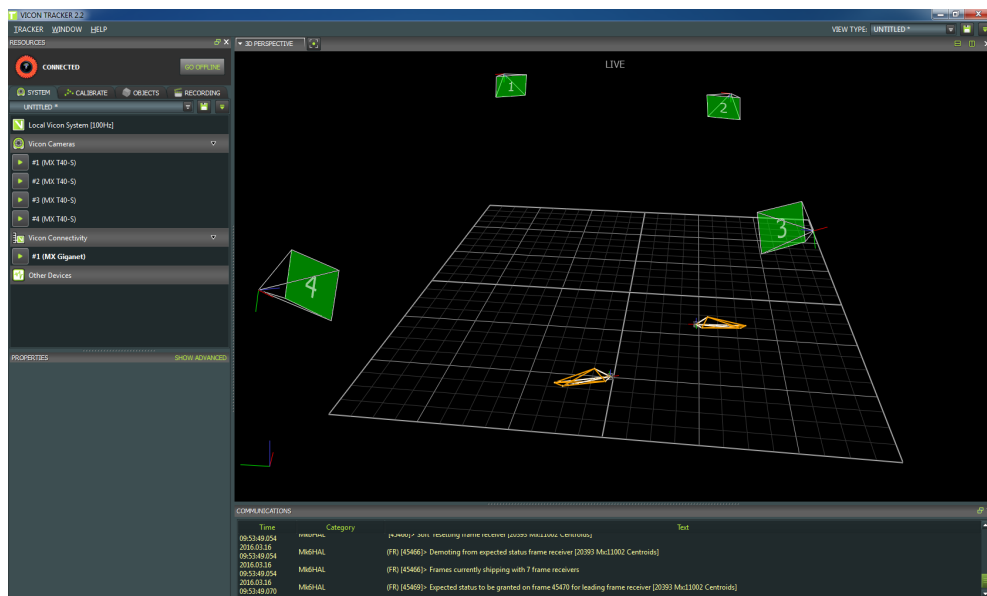
To provide a certain amount of environment perception to the youBots, one Hokuyo URG-04LX-UG01 laser scanner (cf. figure 3.3(a)) is mounted at the front and back of each youBot respectively. These laser scanners are connected to the youBot on-board PC through the USB port, and perform distance measurements in a plane parallel to the ground. Each laser scanner performs 683 distance measurements for an angle of  $240^\circ$ , and provides the results at a rate of 10 Hz with an accuracy of  $\pm 30$  mm [43].



(a) Hokuyo URG-04LX-UG01 laser scanner



(b) Vicon T40S camera

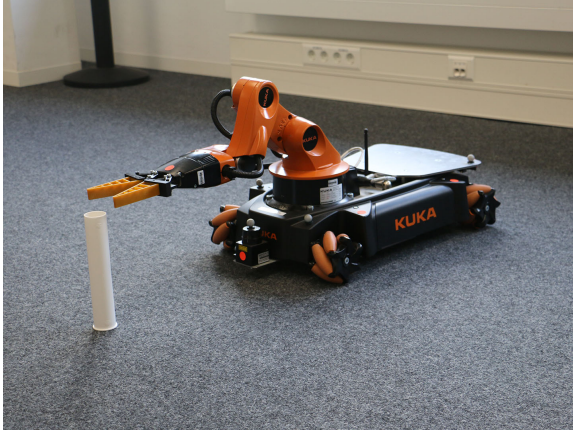


(c) Vicon Tracker software

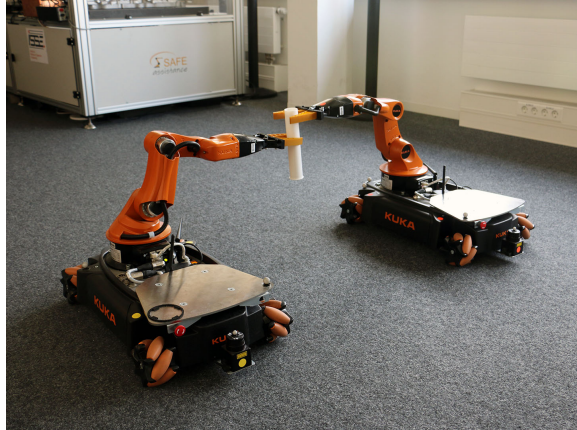
Figure 3.3: The sensors used in the case study

### 3.2.2 External Optical Tracking – Vicon T-Series

In addition to the laser scanners mounted on the youBot providing local perception, a Vicon tracking system for absolute positioning is used. The Vicon system used at the Institute for Software and Systems Engineering at the University of Augsburg consists of up to eight *T40S* cameras (cf. figure 3.3(b)). Each camera contains a ring of infrared LEDs and senses reflections caused by objects in the work-space with a resolution of 4 Megapixels. These cameras are placed in the corners and at the sides of the work space, covering each relevant area with at least two



(a) Picking up the baton



(b) Handing over to the second youBot

Figure 3.4: Application example: Handover in motion

cameras. The cameras are connected and powered through a *MX Giganet* that forwards the sensed data to a PC system running the *Vicon Tracker* software [102].

Objects to be tracked have to be equipped with a unique constellation of infrared retro-reflective markers. In the *Tracker* software (cf. figure 3.3(c)), these three-dimensional marker layouts can be defined as so-called *Subjects* that are then identified and tracked. For tracking, three markers of an object have to be detected by at least two cameras, however to help against occlusions more markers and cameras are helpful. After calibration, the *Vicon* system offers a position reference common to all used robots, relative to which positions and orientations of the *Subjects* are provided at update rates up to 515 Hz.

### 3.3 Application Examples

Using the introduced actuators and sensors, two application examples are used throughout this thesis, spanning from physical interaction between robots defined in advance to more dynamic, user-influenced reactive behavior.

#### 3.3.1 Handover in Motion

The first application example focuses on the interaction of two youBots. One youBot picks up a baton placed in a predefined area of the room, and subsequently takes it to a handover area where it forwards it to the second youBot while both youBots are in motion.

For this example, two youBots and their front laser scanners are used, along with the *Vicon* tracking system. Both youBots are equipped with *Vicon* markers, while the baton is detected using the on-board sensors. Figure 3.4 shows the two parts of the application, first picking up the baton and subsequently handing it over to the second youBot.

The application example can be separated into three phases, each with its own challenges. In the first phase, the baton is picked up, using one youBot platform, arm and gripper, as well as the laser scanner. As the baton is not equipped with markers, its position is not automatically provided by the Vicon system. Thus, a simple form of object detection using the laser scanner is required, searching for poles of a given diameter in a defined area. After driving close to the baton using the youBot platform, a grasp has to be performed. Due to the kinematic structure of the youBot arm, not every grasping position is reachable using the arm alone. Thus, the youBot has to be handled as a mobile manipulator, resolving the redundancy present in the eight degrees of freedom system and finding a solution how the kinematically restricted arm can still grasp the baton. Motions in the reachable space of the youBot arm have to be planned and executed that result in reliably grasping and picking up of the baton.

In the second phase, the youBot moves over to the handover area and synchronizes its motion with the second youBot. In this phase, the two youBot platforms are affected, and the Vicon tracking system is used as a position reference. This requires the youBot platforms to successfully and precisely navigate in the Vicon reference system to reach the handover area. To simplify the handover, the youBot platforms then drive in parallel, keeping their distance constant to allow the arms to perform the interaction as if the youBots were at rest. Therefore, one youBot has to know the position of the other, and be able to drive next to it with sufficient precision to allow the baton to be transferred.

The third phase contains the actual handover. There, the two youBot platforms drive in parallel, while the arms and grippers are used to transfer the baton. Again, the Vicon tracking system is used to ensure parallel motion of the platforms. To successfully perform the handover, the motions and gripper actions of the two youBots have to be synchronized: The first youBot may only release the baton after the second youBot has grasped it, and this gripper sequence may only be executed once both arms are in suitable positions. Additionally, time constraints exist for the handover process, limited by the distance available for parallel motion of the platforms.

### 3.3.2 Gesture Control of a Quadcopter

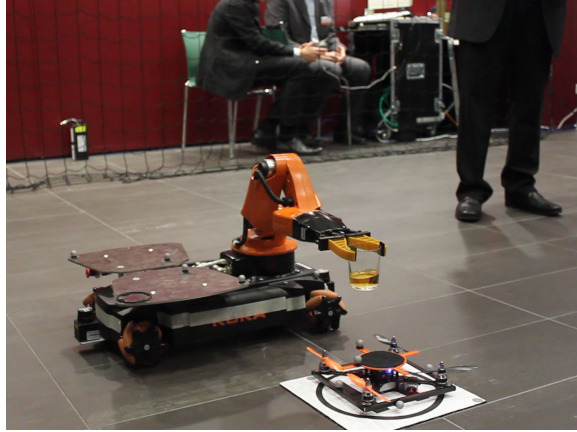
In contrast to the predefined workflow given in the first application, the second example focuses on reactive behavior and user interaction. There, an operator uses a wand (a wooden stick with Vicon markers) as a passive pointing device to trigger certain actions of a quadcopter, to guide the quadcopter along arbitrary paths and to make a youBot place a filled glass on top of the quadcopter.

For this example, a quadcopter and a youBot are used. As a position reference, the Vicon tracking system is used, treating the youBot, the quadcopter, the pointing device and two marked landing zones as tracked *Subjects*. The wand is equipped with four markers in an asymmetric constellation, defining a pointing direction (*forward*) as well as one prominent





(a) Guiding the quadcopter



(b) User-triggered quadcopter-youBot interaction

Figure 3.5: Application example: Gesture control of the quadcopter

direction (*up*). Using this wand, a set of different gestures are defined to trigger behavior of the actuators:

**Activation** The prominent marker denoting the *up* direction is used to activate the wand: If the marker points downwards, the wand is seen as inactive, so its position and motion is ignored. If however the marker points upwards, the wand is active and can trigger actions of the quadcopter or youBot.

**Starting and Landing** If the wand is active for a short time while the quadcopter is landed, the quadcopter is started and flies to the position denoted by the wand tip. If the wand is activated while its tip is in immediate proximity of the landing zone, the quadcopter returns and lands.

**Guiding** When the quadcopter is started, it can be guided using the wand. Whenever the wand is activated, a position 50 cm in front of the wand tip is given as a goal position for the quadcopter. Moving the wand while keeping it active leads to continuous guiding of the quadcopter, while disabling the wand and re-activating it in another position leads to a linear motion to the new position.

**Interaction** When the quadcopter is landed, the youBot can be activated by pointing at it when activating the wand. Starting the quadcopter while the youBot is active triggers the interaction. The quadcopter flies to the transfer point, where the youBot places a glass on top. Then, the youBot retreats and the quadcopter can be started carrying the glass using the usual starting gesture.

In addition to challenges seen in the first application, this example adds three further aspects. As a main difference, this workflow is not completely predetermined, but the order of

actions depends on the behavior of the operator and sensor data. Thus, it cannot be given as a mere sequence, but rather includes case distinctions and reactive behavior, leading to other programming concepts when modeling the application. As a second aspect, the detection and identification of the wand gestures has to be implemented. Therefore, geometric conditions when the wand is active or when it points at the youBot have to be modeled. As a third challenge, the quadcopter as a flying robot (with an internal position estimation even worse than that of the youBot platform) has to be stabilized through Vicon tracking, and integrated into the approach so that it supports the same types of motions as other mobile robots.

The following chapters 4 to 9 describe the concepts required to cleanly model and implement the two application examples, and chapter 10 describes the actual implementation of the examples along with experimental results.



## PHYSICAL OBJECTS AND APPLICATION GEOMETRY

The first step when programming an application for a mobile robot is to define the application geometry, consisting of the relevant (physical) objects involved, as well as their (geometric) relationship. To do this in an object-oriented fashion, both physical objects and relationships are modeled as software objects, which reference each other and thus result in an object structure.

In this context, Requirement 1 (cf. section 2.2) demands the possibility to specify the devices and manipulated objects in Cartesian space, along with further relevant positions. For the proposed software framework, the aforementioned requirement has been refined into a set of design goals to allow flexible programming of robot application. First, it has to be possible to label and describe relevant positions in space, which may for example belong to devices or objects, and which may be linked to or depend on each other:

**Design Goal 4.1.** *Model relevant features in space, along with their conceptual relationships.*

For these features, their position relative to other features is an interesting detail that should be provided. In addition to a static position, some of these features can move and thus also have a relevant velocity. This is especially important for mobile robots that move in their environment and also have to model this aspect to estimate or predict relative motions as well as future positions. The velocity thus has to be managed, and must be available when searching for the velocity of one feature relative to another:

**Design Goal 4.2.** *Model and derive positions and velocities of relevant features.*

Apart from the relevant features that correspond to real-world aspects, it is also required to describe further positions for temporary use. These may describe places to look at, robot

trajectories that might be executed in the future, or goal positions from motion specifications, and provide an effective means to share plans and expected outcomes between cooperating robots:

**Design Goal 4.3.** *Reference further places in space, either to talk about positions, or as hypothetical or desired positions of features.*

While reality defines the ground truth about the world, robots may only have a partial, incorrect or noisy view on the environment. Multiple robots used together may additionally have different information about the world, which may lead to disagreeing views of the world. To reliably support the cooperation of multiple mobile robots, the software framework has to be able to cope with this, and allow applications to access the information relevant to the current context:

**Design Goal 4.4.** *Model different views of the world.*

Apart from views for the knowledge of different robots, the views can also be used to describe hypothetical situations: One view may describe the situation a robot is trying to achieve, while another view is used for an intermediate situation a motion planner wants to check for collisions.

Instead of composing the world of individual geometric features, object-orientation promotes the use of domain objects. Thus, actuators and manipulated objects are modeled as software objects knowing their important features. They describe the linkage of rigid bodies and are usable for geometry description even if no concrete actuator instance can be controlled in the given context (e.g. when the Device belongs to another system and thus does not provide run-time information in the current execution environment, cf. section 2.3):

**Design Goal 4.5.** *Define the geometry of physical objects and actuators, independent from their execution environment.*

Keeping the application definition independent from the execution environments is especially important with distributed robots, where the same domain objects can be known to different robots and be used in different execution environments. Apart from the objects and actuators used, their application-specific placement in the environment is important. This placement may be known (“On the top left corner of the desk”) or unknown (“Somewhere in the cupboard”), both of which being important information to model:

**Design Goal 4.6.** *Define the relationship of physical objects, being known or unknown.*

Using mobile robots in a less structured environment<sup>1</sup>, positions of most objects are unknown or at least only rough approximations, so modeling unknown relationships plays an important role. Finally, the initial environment set-up should not be an integral part of an application workflow, but rather be defined independently for use with further applications:

---

<sup>1</sup>Throughout this thesis, “less structured environments” is used for environments where in contrast to industrial robotics the exact positions of robots and work pieces are not exactly defined and ensured, however the objects and robots to be expected are known beforehand.

**Design Goal 4.7.** *Model geometry independent from application workflow.*

The following sections go into detail how positions and geometric relationships in Cartesian space are expressed (section 4.1), and how physical objects (section 4.2) and their place in the world (section 4.3) are modeled. These sections are based on the work of Angerer [2], however with modifications required for mobile robot applications. The chapter is concluded by a description of these modifications (section 4.4), along with other approaches of modeling the geometry of a robotic application (section 4.5).

## 4.1 Representing Cartesian Space

For most robot tasks, the position of robots and relevant items is important. These positions are typically described in Cartesian space, based on mathematical foundations described in section 4.1.1. These foundations allow to define relevant features in space, together with their relationships (section 4.1.2), and to easily define other positions (section 4.1.3), orientations (section 4.1.4) or velocities (section 4.1.5) relevant to the application.

### 4.1.1 Mathematical Representation – Vectors, Rotations and Matrices

The following description of concepts in Cartesian space is based on the established mathematical formalism presented by Waldron et al. [105, chapter 1.2], as similarly described by Angerer [2, chapter 7]. This section introduces the aspects required to understand the concepts used in this thesis, while the aforementioned sources are advised for further reading.

Cartesian space is represented as a three-dimensional space,  $\mathbb{R}^3$ . In this space, coordinate reference frames (further called frames) can be defined as an origin point together with three orthonormal vectors forming a basis and thus defining coordinate axes. Based on a frame  $f$ , another position  $v$  can be described, expressing the displacement from the origin point to the position through a linear combination of the three basis vectors. This displacement is expressed through a vector  ${}^f_f\vec{v} \in \mathbb{R}^3$ , giving the displacement along the x, y, and z axis relative to and expressed in the frame  $f$ .<sup>2</sup>

$${}^f_f\vec{v} = \begin{pmatrix} {}^f_f v_x & {}^f_f v_y & {}^f_f v_z \end{pmatrix}^T$$

A further frame  $g$  can be described relative to  $f$  by giving the displacement of the origin of  $g$  expressed in  $f$ , along with the direction of the basis vectors of  $g$  expressed in the basis of  $f$ . Here, the displacement is again expressed as a vector  ${}^f_f\vec{p}_g$ , while the new basis is written as a matrix  ${}^f_f R_g \in \mathbb{R}^{3 \times 3}$ , expressing the direction of the x, y, and z axis of  $g$  in the basis of  $f$  as first,

<sup>2</sup>In the referenced work by Waldron et al. [105],  ${}^f\vec{v}_g$  is used instead of  ${}^f_f\vec{v}_g$  to describe this concept. However, in sections 4.1.3 and 4.1.4 positions are defined based on the reference point to work with and the definition of the x, y and z axis as two independent concepts that are given as left indices, motivating to use the same formalism here for consistency reasons.

second and third column.

$${}^fR_g = \begin{pmatrix} \vec{x} & \vec{y} & \vec{z} \end{pmatrix}$$

The complete transformation of  $g$  relative to  $f$  can be expressed as a single homogeneous transformation matrix  ${}^fT_g \in \mathbb{R}^{4 \times 4}$ :

$${}^fT_g = \begin{pmatrix} {}^fR_g & {}^f\vec{p}_g \\ 0 & 1 \end{pmatrix}$$

To transform a position  ${}^g\vec{v}$  defined relative to frame  $g$  into frame  $f$ , a matrix multiplication can be applied after extending the vector by one dimension filled with the value 1 (known as homogeneous coordinates):

$$\begin{pmatrix} {}^f\vec{v} \\ 1 \end{pmatrix} = {}^fT_g \cdot \begin{pmatrix} {}^g\vec{v} \\ 1 \end{pmatrix}$$

Similarly, the transformation  ${}^gT_h$  of a frame  $h$  expressed relative to  $g$  can be converted into a transformation relative to frame  $f$  using a matrix multiplication:

$${}^fT_h = {}^fT_g \cdot {}^gT_h$$

Navigating a transformation in the opposite direction can be achieved through matrix inversion:

$${}^gT_f = {}^fT_g^{-1}$$

This way, it becomes possible to work with chains of frames, where each one is defined relative to the previous one, and to convert positions that are defined based on any part of the chain.

Apart from directly giving the rotation matrix, rotations can also be specified using other formalisms. Typical ones include Euler angles, an axis-angle representation or quaternions. In the case of Euler angles, rotations are given as three successive rotations around the coordinate axes, thus describing arbitrary rotations with 3 scalar values. In the Robotics API, Euler angles are used, consisting of a rotation around the Z axis, followed by one around the new (rotated) Y axis, followed by one around the new X axis. Following the convention used by KUKA, the three angles are called  $A$  (for Z rotation),  $B$  (for Y) and  $C$  (for X). For given  $A$ ,  $B$ , and  $C$  angles, a rotation matrix can be calculated by multiplying three individual rotation matrices for the corresponding rotations around Z, Y and X. Conversely, for a given rotation matrix, values for  $A$ ,  $B$ , and  $C$  can be found. Another way to describe rotations is through the axis-angle formalism. There, the rotation is defined through a rotation axis given as a vector, together with a scalar angle specifying how far to rotate around the axis. It is again possible to convert this representation from and to a rotation matrix, so it can be used whenever it simplifies specifying the desired rotation. Finally, unit quaternions can be used to specify rotations through four values (cf. [105]), which can again be converted into the other representations.

To describe motions of frames, the concept of a twist is introduced. The twist consists of two vectors in  $\mathbb{R}^3$ , one for the translational velocity and one for the rotational velocity of a

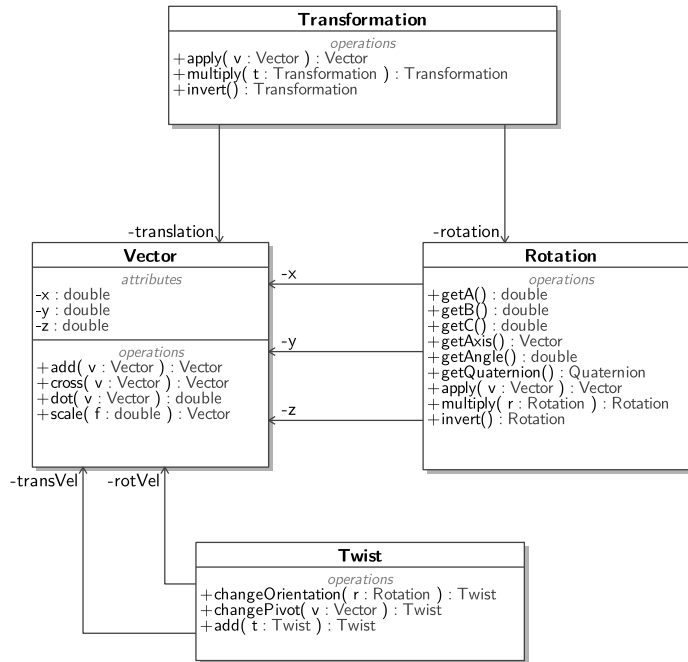


Figure 4.1: Primitive types in Cartesian space

frame. The vector for the translational velocity describes how the displacement relative to another frame changes, expressed in the basis chosen as an orientation. The direction of the vector gives the direction of the motion, while the length specifies the speed. The rotational velocity describes the change of the basis vectors of the moving frame relative to the reference frame, again expressed in the chosen basis. Here, the direction of the vector defines an axis of rotation (similar to the axis-angle representation of rotations), while the length specifies the speed. Additionally, the rotational velocity can contribute to the translational motion, because it describes a rotation around a specified point. If this point does not coincide with the moving frame, rotational velocities result in circular motion of the moving frame around the given center, displaced by the given translational velocity.

To use these concepts in the Robotics API, corresponding classes are provided (cf. figure 4.1). **Vectors** are used to represent the coordinates of points in 3D space. They provide calculation methods for component-wise adding and scaling, together with the cross and dot products defined for three-dimensional vectors. A **Rotation** consists of three Vectors for the three coordinate axes, and provides access to different representations, especially Euler angles (A, B, C) and the axis-angle representation. Additionally, they allow to rotate a given Vector (*apply*), to create an inverted Rotation and to multiply with another Rotation, representing the sequential execution of both Rotations. **Transformations** consist of a Vector describing the position, together with a Rotation for the orientation. They can again be inverted, multiplied with another Transformation and applied to a Vector. **Twists** use two Vectors to express the translational and rotational

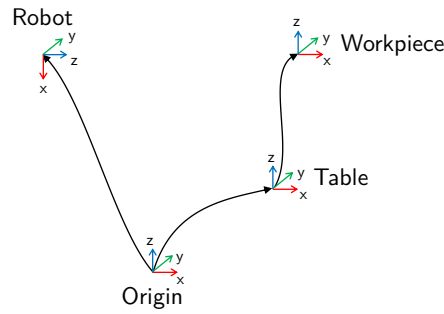


Figure 4.2: Example set-up with four Frames and their Relations

velocity, and can be transformed to be expressed in another orientation or using another rotation center (pivot).

However, these classes only hold numbers, but do not give any metadata about the used reference frame or coordinate system. Consequently, the calculation methods can be applied to values that are not expressed in compatible bases, yielding results without correspondence in Cartesian space. To solve this problem, further data types are introduced that work on top of these primitive types and include semantic information as well as validity checks. These types are introduced in the following sections.

#### 4.1.2 Frame and Relation – Named Places and their Relationships

To talk about Cartesian space in robot applications, the concept of **Frames** is introduced, following design goal 4.1. A Frame represents a (named) place in space, indicating a position (in three-dimensional space) as well as an orientation (defining three orthonormal coordinate axes of the space). It can be used to reference a corresponding geometric feature in Cartesian space, but also as a basis to define new poses relative to it.

The relationship between two Frames is defined in a **Relation**. Relations define a logical link between the frames, and also describe the durability of the link. If the displacement of the link is known, it is given as a Transformation describing the translation and change of orientation between the two Frames. Note that a Frame per se does not know its absolute position in space – the position is only defined through its Relations. The structure of Frames and Relations forms an undirected multi-graph<sup>3</sup>, which allows to navigate between different Frames if a sequence of Relations exists that form a path between the two Frames.

Figure 4.2 gives an example of Frames and Relations. Frames are depicted as named groups of arrows, which give the position as the intersection between the three arrows. The X direction of the corresponding orientation is given by the red arrow, while green stands for Y and blue for the Z direction. Relations are shown as black arrows. The example shows Frames for a *Table* and

<sup>3</sup>In the implementation, Relations have a direction that tells how to interpret the stored Transformation, however the Relation can also be navigated in the opposite direction.

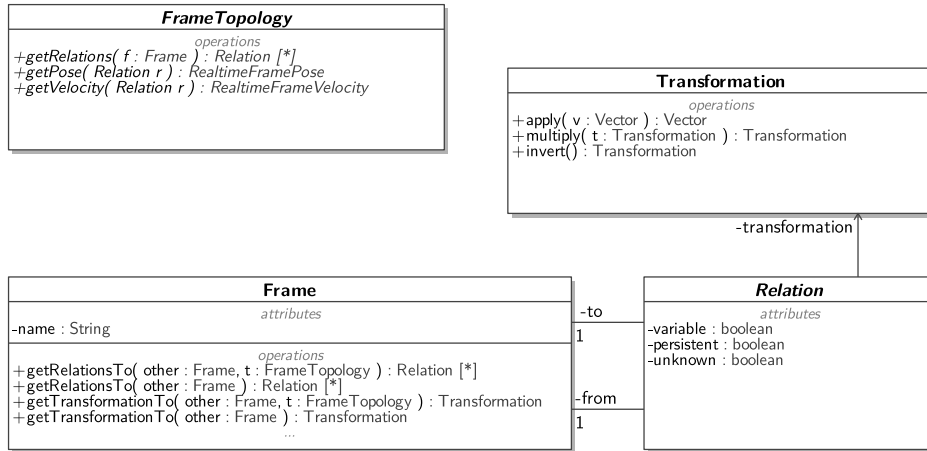


Figure 4.3: Frames and Relations describing geometry

a *Robot* which are defined relative to an *Origin* frame, along with a *Workpiece* that is defined relative to the *Table* it is placed on. The Relation to the *Workpiece* can be described as a pure translation in X and Z direction, while the Transformation between *Origin* and *Robot* includes a rotation around the Y axis.

When describing more complex mechanisms such as complete robot arms, not all relationships remain constant all the time. Thus, different types of Relations are required. These have to be differentiated with respect to three different properties:

**Constant / Variable** Relations can either have a variable or constant Transformation. While static rigid connections have a constant Transformation and are thus called constant, other connections such as the connection between two robot Links (represented by a Joint) can change (or even be controlled) over time and are thus called variable.

**Persistent / Transient** Some Relations will typically remain present for the entire run time of an application, while others are created and removed during operation. A robot arm screwed onto a mobile platform will likely remain there and thus uses a persistent relation, whereas a work piece placed on a mobile platform can easily be grasped and thus uses a transient relation.

**Known / Unknown** While some Transformations are known exactly (e.g. the size of robot Links from CAD data), for others no exact Transformation is known. This is especially common with mobile robots with their inherent imprecision through locomotion, as well as with the less structured environments mobile robots are typically used in. In these cases, unknown Relations can be used that cannot tell the exact Transformation themselves, but may give constraints for the possible Transformations. Being neither exactly known nor totally unknown, some Relations may only provide the position with a certain amount of uncertainty. These could also be modeled as known Relations storing their uncertainty

as a covariance matrix, however such Relations are not relevant in the case study and are thus not further considered in this thesis.

Frames and Relations together describe the geometric set-up of the environment, called **World model**. It forms the basis for robot motions performed in Cartesian space, as well as for reasoning about possible actions and their effects on the environment. In software, the World model is expressed through instances of the classes Frame and Relation, together with Transformations describing the displacement (cf. figure 4.3).

To find the Transformation between two given Frames as requested in design goal 4.2, the World model is traversed in a breadth-first-search starting at the start Frame and goal Frame, searching for a set of Relations that link the two Frames. On the way, Relations are ignored if they do not provide a Transformation (e.g. if uncertain or unknown) or are unavailable in the given context (cf. chapter 9). Having the list of Relations, their Transformations can be multiplied (possibly inverted if the Relation points in the opposite direction) to calculate the Transformation from the start to the goal Frame.

Apart from the World model defined through the Relations known by the Frames and the Transformations in the Relations, further geometric set-ups can be given through a **Frame-Topology** following design goal 4.4. It defines which Relations can be reached from Frames and which value to assume as Transformation of Relations. Using FrameTopologies allows to restrict the graph search between given Frames to a certain type of relations, or to find Transformations assuming a certain environment state. The Relations can for example be limited to constant Relations, to exclude unknown Relations or to perform the search using information available on a given execution environment. Giving the environment state allows to perform geometric calculations under given assumptions, allowing to answer the question where an object would be if its carrying robot was in a given configuration.

For example, one FrameTopology can be used that describes one robot's view of the world. It includes all the Relations that are known or can be observed by the robot through sensors, and assigns them the corresponding sensor measurements. This FrameTopology can then be used to reason about the robot's knowledge, or to define a goal for the robot (causing an error if the given goal is not known to the robot). A further FrameTopology describes all Frames that are linked to each other in constant and persistent way. All computation results calculated in this topology can then be used persistently, as they will not change unless Relations are removed. A third FrameTopology describes the situation a robot expects when a task has been executed. It contains all the Relations from the World model, however assigns values to them that describe the Transformations that are expected to be valid after execution (e.g. the goal position). Using this topology, a collision check can be performed, or next steps to be executed can be planned.



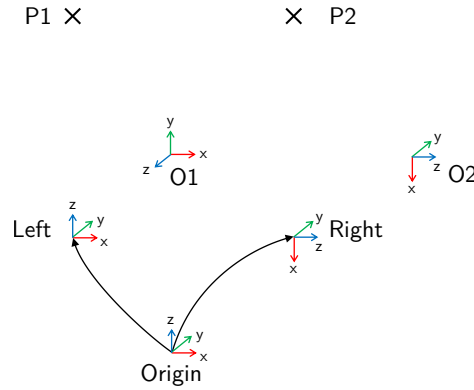


Figure 4.4: Example set-up with different Points and Orientations in Cartesian space

### 4.1.3 Point and Orientation – Positions and Coordinate Axes

In general, a Frame describes a position in space and the direction of coordinate axes. Omitting the coordinate axes, the position of a Frame, but also further positions that do not belong to a named entity can be described as **Points**. Similarly, an **Orientation** allows to represent a given triple of coordinate axes. An Orientation is defined relative to a Frame, optionally with a given Rotation to be applied to the Frame's Orientation to reach the desired triple of coordinate axes. Similarly, it is possible to construct a Point by giving a Vector as a displacement of a Frame's position, interpreted in a defined Orientation.

In figure 4.4, three Frames are shown, along with two further Orientations (*O1* and *O2*) and two Points (*P1* and *P2*). The Orientation *O2* can be described in different ways, depending on the used reference Frame: It either consists of an identity Rotation using *Right* as reference Frame, or of a  $90^\circ$  rotation around the Y axis when using *Left* or *Origin* as a reference. Similarly, *O1* can be defined through a  $90^\circ$  rotation around the X axis using *Left* or *Origin* as reference, or through a more complex rotation based on *Right*. The Point *P1* can be described by a Z translation using *Left* as a reference Frame and Orientation. However, the same Point can also be described using *Right* as reference and Orientation and a translation in negative X and Z direction. Using a different reference and Orientation, *P1* is also expressible as a Y translation relative to *Left*, using *O1* as an Orientation. The different choices of reference Frame and Orientation have an influence on the run-time behavior of Points: If the corresponding Frame moves (caused by a variable Relation), the Point moves along and can thus still describe the intended feature or position.

From a software point of view, Points and Orientations are modeled by the corresponding classes, as shown in figure 4.5. Using the Transformations given in the World model or a provided FrameTopology, Points and Orientations can be converted to use other reference Frames. For an Orientation  $o$  with reference Frame  $r$  and rotation matrix  ${}_rO$ , the reference Frame can be changed to  $r'$  by calculating the Transformation  ${}_{r'}T_r$  between the new reference Frame  $r'$  and

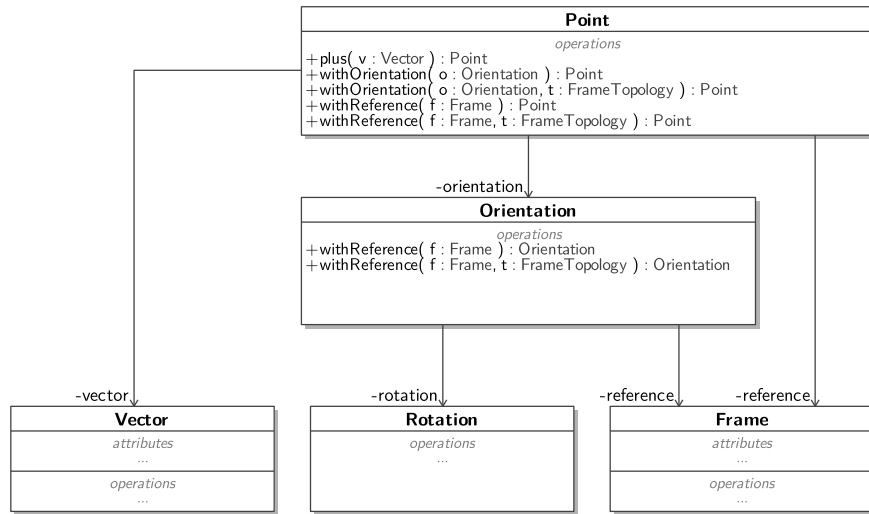


Figure 4.5: Software model for Points and Orientations

the previous one  $r$ , reducing it to a Rotation matrix  ${}^{r'}_r R_r$  and applying it to the Rotation.

$${}^{r'}_r O = {}^{r'}_r R_r \cdot {}^r O$$

For a Point  $p$  expressed in reference Frame  $r$  and an Orientation  $o$  that matches the Orientation of  $r$ , the Vector  ${}^o_r \vec{p}$  can similarly be transformed to reference  $r'$ , while changing the Orientation to the Orientation of  $r'$ , called  $o'$ :

$${}^{o'}_r \vec{p} = {}^{r'}_r T_r \cdot {}^o_r \vec{p}$$

To change the Orientation of a Point from  $o$  to  $o'$ , the Rotation matrix  ${}^{o'}_o R_o$  from  $o'$  to  $o$  is calculated and left-applied to the Vector:

$${}^{o'}_r \vec{p} = {}^{o'}_o R_o \cdot {}^o_r \vec{p}$$

This calculation is also required to change the reference Frame of a Point that uses an Orientation different from its reference Frame. Furthermore, based on a given Point, a new Point with an additional displacement can be constructed through the *plus* method. When a Frame-Topology is provided with the conversion methods, its values are used to compute the required Transformations.

#### 4.1.4 Pose and FramePose – Positions with Orientation

When talking about positions in space including Orientation, **Poses** become useful. While conceptually similar to a Frame linked by a Relation, a Pose serves as a more light-weight, temporary concept to talk about a place, without the identity and persistence properties of a named place represented by a Frame. As shown in figure 4.6, a Pose is defined by a reference Frame, an Orientation and a Transformation including a Vector for translation and a Rotation

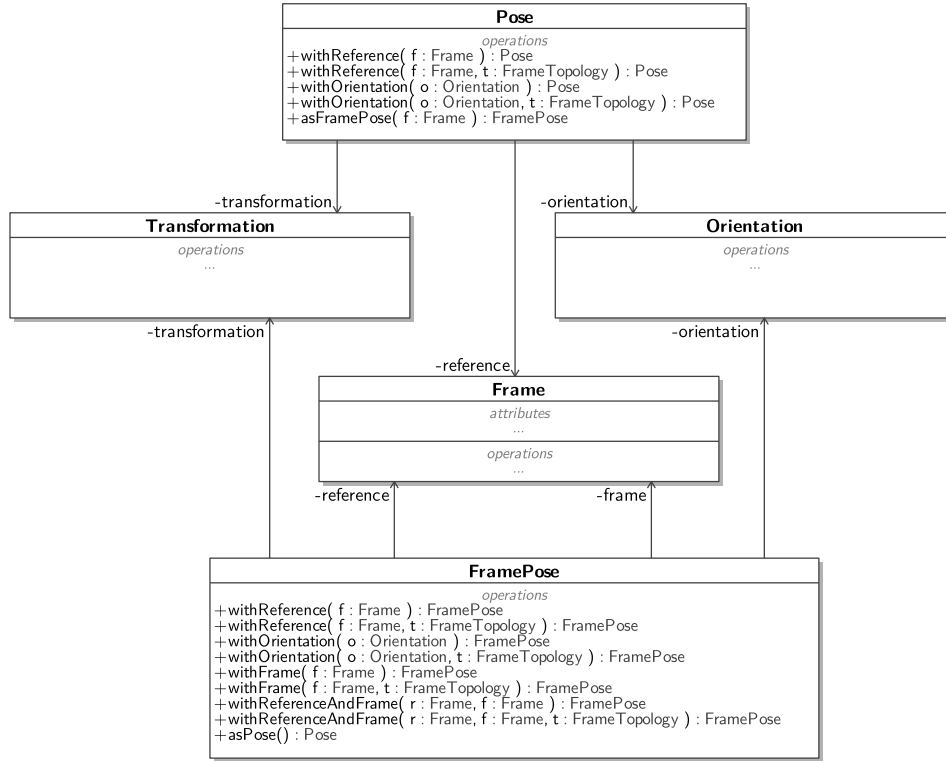


Figure 4.6: Poses and FramePoses in Cartesian space

for the orientation change. It represents the position reached when starting at the reference Frame, and applying the Transformation interpreted in the given Orientation. In many cases, the Orientation is equal to the reference Frame's Orientation (which leads to a behavior similar to that of Relations between Frames), however being able to use a different Orientation greatly simplifies the description of some Poses.

Using Transformations available in the World model or in a given FrameTopology, a Pose  $p$  with reference Frame  $r$ , Orientation  $o$  and Transformation  ${}^o_rP$  provides three helpful operations:

**withOrientation( $o'$ : Orientation)** Computes a new Pose that describes the same place in space, however expressed using another Orientation. Therefore, the Rotation matrix  ${}^{o'}_oR_o$  between the new and old Orientation is computed, extended to a Transformation  ${}^{o'}_oT_o$  by adding an empty translation, and applied to the Transformation  ${}^o_rP$  as a basis change:

$${}^{o'}_rP = {}^{o'}_oT_o^{-1} \cdot {}^o_rP \cdot {}^{o'}_oT_o$$

**withReference( $r'$ : Frame)** Computes a new Pose that describes the same place in space, however using the reference Frame  $r'$ . The implementation therefore finds the Transformation  ${}^{r'}_{r'}T_r$  between the new reference Frame and the old one, and left-applies it to the known Transformation  ${}^o_rP$ :

$${}^{o'}_{r'}P = {}^{r'}_{r'}T_r \cdot {}^o_rP$$

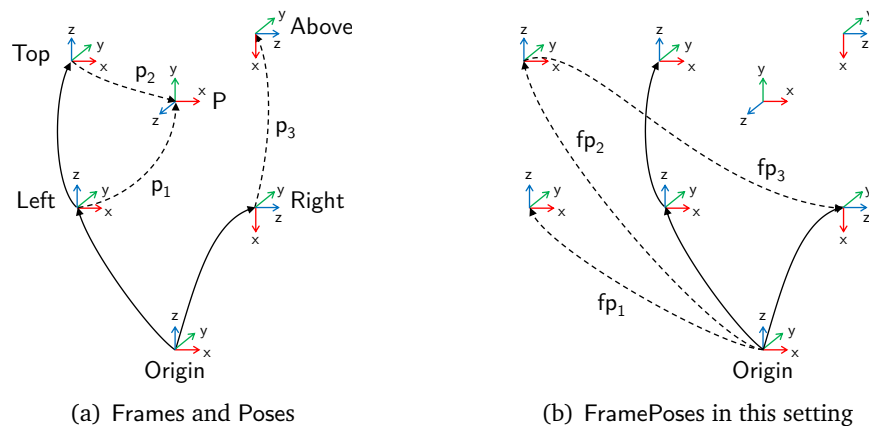


Figure 4.7: Example set-up with different Frames, Poses and FramePoses. Relations are shown as solid arrows, while Poses and FramePoses are dashed.

However, this computation is only valid if the Orientation  $o$  corresponds to the Orientation of the reference Frame  $r$ , and the result has the Orientation  $o'$  of the Frame  $r'$ . If the initial Orientation  $o$  does not fulfill this requirement, it is changed through `withOrientation()` before applying the computation.

In the example of figure 4.7, the position and orientation of  $P$  can be given as a Pose  $p_1$  relative to *Left* with a translation in positive X and Z direction, together with a rotation around the X axis. When changing the Orientation to *Right* using `withOrientation()`, the corresponding Transformation changes into a translation in Z and negative X direction and a rotation is around Z. Using `withReference()`, the reference Frame can for example be changed to *Top*, which leads to Pose  $p_2$  described by a translation with negative Z component. To define the Pose  $p_3$  of *Above* that is above its reference *Right*, *Origin* can be chosen as an Orientation so that the Transformation consists of a translation in Z direction no matter how *Right* is oriented.

Sometimes, a Pose is used to describe a position of a certain Frame, such as a measured, desired or hypothetical position. In this situation, a **FramePose** can be used. In addition to the properties and methods of a Pose, it stores the Frame whose position it describes, and allows to compute the position of other Frames assuming that the described Frame is at the given position. A FramePose  $p$  describing the Frame  $f$  relative to  $r$  thus provides two further operations:

**withFrame( $f'$ : Frame)** Computes a new FramePose that describes the pose of the given new Frame  $f'$  under the assumption that the old Frame  $f$  is at the given FramePose  $p$  with Transformation  ${}^oP_f$ . Computationally, this is performed by right-applying the Transformation  ${}^fT_{f'}$  between the current Frame  $f$  and the new Frame  $f'$  to the current Transformation  ${}^oP_f$ :

$${}^oP_{f'} = {}^oP_f \cdot {}^fT_{f'}$$

This computation requires that the Orientation  $o$  corresponds with the reference Frame  $r$ , and thus uses `withOrientation()` to change it if required.

**withReferenceAndFrame( $r'$ : Frame,  $f'$ : Frame)** Computes a new FramePose describing the pose of the new Frame  $f'$  relative to the new reference Frame  $r'$ , taking the given FramePose  $p$  as granted (i.e. it assumes that  $p$  describes the current Transformation of Frame  $f$  relative to the reference Frame  $r$ ). Computationally, this is a combination of `withFrame()` and `withReference()`, however handling the special case when the path from new reference  $r'$  to new Frame  $f'$  traverses the Pose in the opposite direction as the old reference  $r$  and Frame  $f$ .

In the example of figure 4.7, a FramePose  $fp_1$  describes a hypothetical position and orientation of the Frame *Left* relative to *Origin*. Then, `withFrame()` can be used to compute the FramePose  $fp_2$  of *Top* relative to *Origin*, assuming that – relative to *Origin* – *Left* has the Transformation given by the FramePose. Similarly, `withReferenceAndFrame()` can be used to compute the FramePose of *Top* relative to *Right* assuming the given Transformation between *Left* and *Origin* is correct, leading to the same effect as using `withFrame()` and `withReference()` individually. Additionally, `withReferenceAndFrame()` also allows to compute the FramePose  $fp_3$  of *Right* relative to *Top*. This computation cannot directly be done through `withFrame()` and `withReference()`, because `withFrame()` would lead to a FramePose of *Right* relative to *Origin* that does not contain the given FramePose  $fp_1$ . Instead, the given FramePose first has to be inverted into the FramePose of *Origin* relative to *Left*, before `withFrame()` and `withReference()` are applicable.

For a Pose, the given Orientation must in general be based on a Frame that is connected to its reference Frame. However, sometimes it is easier to describe the Pose from the view of the described orientation. In this case, *null* can be used as Orientation. For the Pose  $p_1$  in figure 4.7, using *null* as an Orientation leads to a Transformation with positive X and Y components. When using a FramePose however, it is possible to directly give the orientation of the described Frame  $f$  as an Orientation. In contrast, the Orientation may not be set to *null* here. Additionally, with FramePoses it is possible to use an Orientation based on a Frame linked to  $f$ , which cannot be expressed using only a Pose.

From a software view, despite similar methods FramePose is not modeled as a specialization of Pose. This is caused by the different semantics of *null* values as an Orientation that prevents full use of FramePoses in places where the software expects Poses, along with different return types for the methods to change reference and orientation. However, `asPose()` and `asFramePose()` allow to convert between the two types while respecting the semantics of *null* values.

#### 4.1.5 Velocity and FrameVelocity – Talking about Motion

When it comes to motion, the concept of a **Velocity** comes into play. It describes a translational and rotational motion in space and is defined through a reference Frame, an Orientation, a

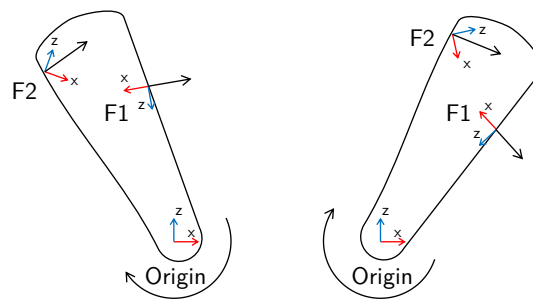


Figure 4.8: Example for different ways of expressing the velocity of a rotating Frame

Pivot point and a Twist. As motions are relative, the Velocity has a reference Frame that is seen as fixed when looking at the moving point. The motion itself is defined through a Twist holding two Vectors, one for the translational velocity (called *Trans*) and one for the angular velocity (called *Rot*). Both Vectors are interpreted in the given Orientation. The angular velocity is assumed to rotate the moving point around the given **Pivot point**. If the Pivot point is not placed at the moving point, the angular velocity causes circular translational motion. In addition to this circular motion, the translational velocity Vector describes the remaining parts of the translational motion.

In figure 4.8, two Frames  $F1$  and  $F2$  are assumed to be mounted on a robot link rotating around *Origin*. Then their instantaneous motion can be described in different ways: Either, the motion is given using *Origin* as reference and Orientation, and  $F1$  or  $F2$  as Pivot points respectively. Then, the Twist is composed of a rotation around the Y axis, together with a translation that is tangential to the circular trajectory the Frame takes. In this representation, a given velocity is only instantaneously valid, but changes once the Frame has moved on. A second way is to give the Velocity using the moving Frame as Orientation and Pivot point. Then the translational velocity of  $F1$  is constant in negative X direction, while  $F2$  has a constant translation in positive X and Z direction. Here again a rotation around the Y axis is present. However, as these Frames are mounted onto the same rotating disc, the most convenient way to model the rotation is to use the rotation center (i.e. *Origin*) as a Pivot point. Then, the Twists of  $F1$  and  $F2$  are equal and can be described as a purely rotational velocity around the Y axis.

For a given Velocity  $v$  with reference Frame  $r$ , Orientation  $o$ , Pivot point  $p$  and Twist  ${}^o_rV^p$ , different aspects can be changed to get a more suitable representation (cf. figure 4.9). Therefore, operations are available, some of which however cannot work on the Velocity alone, but sometimes also require information about the current Pose of the moving Frame. As an alternative to the Pose, a FrameTopology can be given that includes the Pose and other aspects of the assumed situation.

**withReference( $r'$ : Frame)** Computes a new Velocity that represents the same motion of the moving Frame, however expressed relative to another reference Frame  $r'$  (that may itself

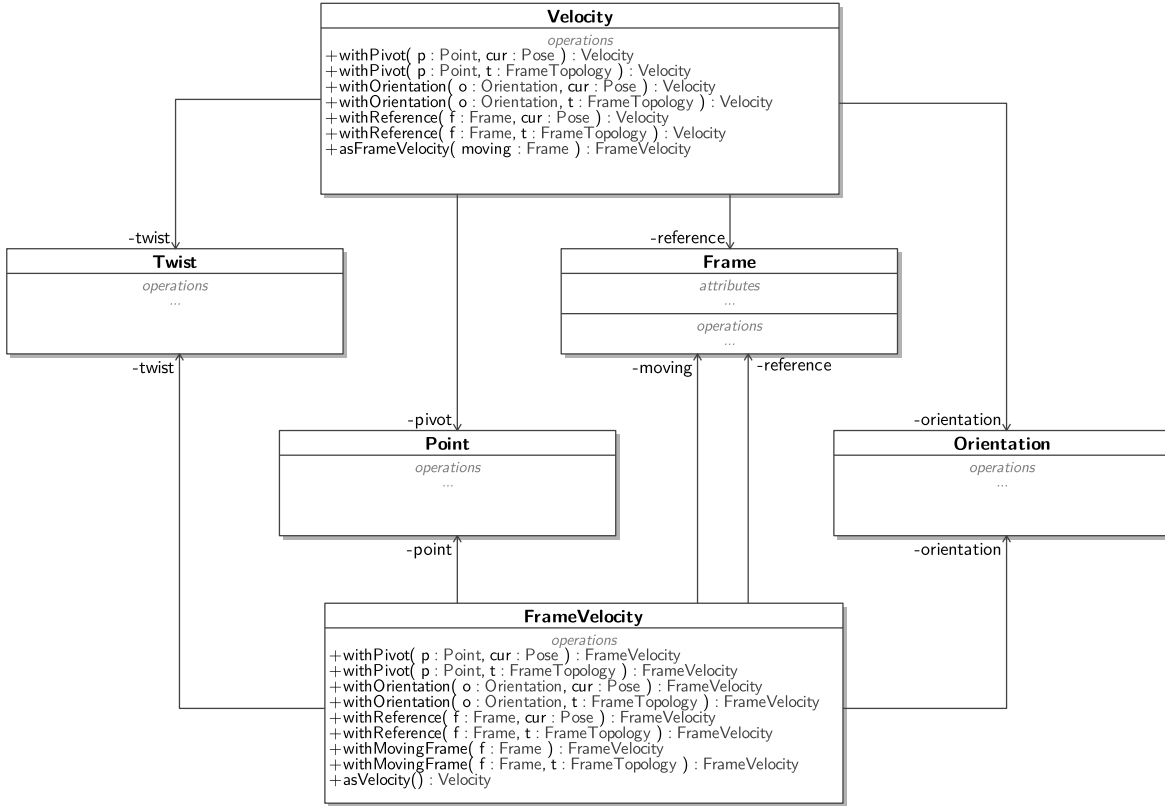


Figure 4.9: Velocities in Cartesian space

be moving). To calculate the new Twist, the Twist  ${}^r_r V_r$  of the current reference Frame relative to the new reference Frame is calculated, and added to the Twist of the current Velocity (both using the current reference Frame  $r$  as Pivot point and Orientation).

$$Trans({}^r_r V^r) = Trans({}^r_r V_r) + Trans({}^r_r V^r), \quad Rot({}^r_r V^r) = Rot({}^r_r V_r) + Rot({}^r_r V^r)$$

The result is then converted to use the Orientation of the new reference Frame using `withOrientation()` and to the previous Pivot point using `withPivot()`.

**withOrientation( $o'$ : Orientation,  $p$ : Pose)** Computes a new Velocity that describes the same motion, however expressed from the view of another Orientation. First, the Rotation between the old and the new Orientation is calculated. If the old and new Orientation are on different sides of this Velocity, the Transformation is calculated using the given Pose, otherwise the World model or given FrameTopology is used. Then, both angular and translational velocity of the Twist are transformed using the calculated Rotation to find the new Twist.

$$Trans({}^{o'}_r V^p) = {}^{o'}_o R_o \cdot Trans({}^o_r V^p), \quad Rot({}^{o'}_r V^p) = {}^{o'}_o R_o \cdot Rot({}^o_r V^p)$$

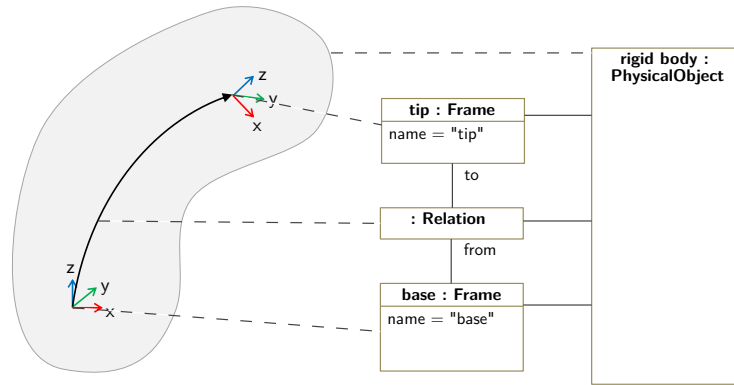


Figure 4.10: A rigid body and its object-oriented representation

**withPivot( $p'$ : Point)** Computes a Velocity that expresses the same motion using another Pivot point. First, the translation from the old Pivot point to the new one is calculated, expressed in the given Orientation. Then it can be used to calculate the new linear velocity as sum of the old one and the cross product of the Pivot point translation and the angular velocity. The angular velocity itself remains unchanged.

$$Trans({}_r^o V^{p'}) = Trans({}_r^o V^p) + Trans({}_p^o T_{p'}) \times Rot({}_r^o V^p), \quad Rot({}_r^o V^{p'}) = Rot({}_r^o V^p)$$

Setting Pivot point or Orientation of a Velocity to *null* means that the moving Frame is to be used as Pivot point or Orientation, and `withPivot()` and `withOrientation()` can be used to convert from and to this representation. If a Velocity specifies the moving Frame  $f$  (and is thus a **FrameVelocity**), Pivot point and Orientation may not be *null* any more, but one further operation becomes available:

**withFrame( $f'$ : Frame)** Creates a new FrameVelocity that expresses the Twist  ${}_r^o V_{f'}^p$  of the new Frame  $f'$ , given that the old Frame  $f$  is moving with the Velocity's Twist  ${}_r^o V_f^p$ . This new Twist can simply be calculated by adding the current Twist and the Velocity of the new Frame relative to the old Frame, expressed in the Velocity's Orientation and Pivot point.

$$Trans({}_r^o V_{f'}^p) = Trans({}_r^o V_f^p) + Trans({}_f^o V_{f'}^p), \quad Rot({}_r^o V_{f'}^p) = Rot({}_r^o V_f^p) + Rot({}_f^o V_{f'}^p)$$

Using these operations, positions and velocities expressed in arbitrary reference Frames and Orientations can be accessed and transformed in a consistent way, allowing to model the motion of mobile robots, as well as their view of the world while in motion. This enables programmers to specify tasks in the most suitable representation, while the software framework is able to use the representation required for task execution.



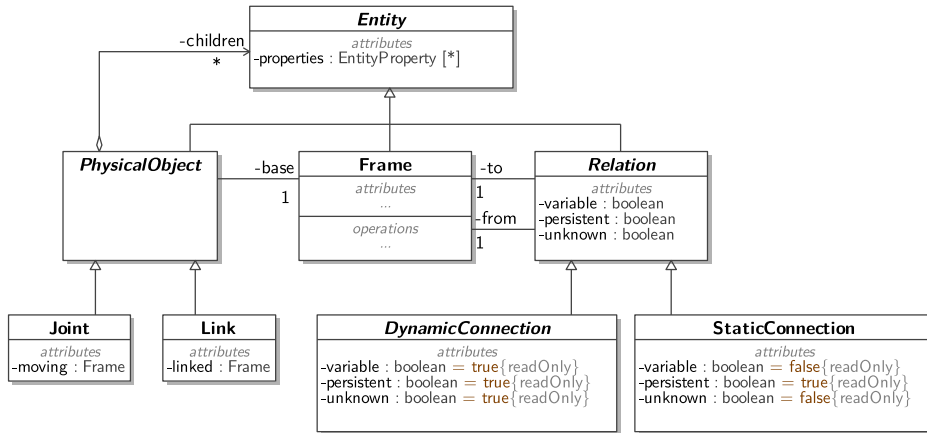


Figure 4.11: Concepts related to PhysicalObjects

## 4.2 Modeling Physical Objects and their Geometric Features

In addition to Frames representing geometric locations, objects present in the world play an important role for robot applications. Relevant objects of a robotic cell are modeled as **PhysicalObjects** and composed of (one or multiple linked) rigid bodies<sup>4</sup> as introduced by Hoffmann [39]. Common objects such as robots can often be used from a library, while application specific objects such as special work pieces can be defined, instantiated and used in an application.

Talking about geometry, physical objects define relevant positions as Frames. Each object has at least one Frame for each rigid body it contains, but can also have multiple to provide access to relevant geometric features. Figure 4.10 gives an example of a **PhysicalObject** with two named places (Frames) and their displacement (Relation).

If a Relation connects two Frames belonging to the same **PhysicalObject**, it describes the displacement of its rigid bodies or of an important geometric feature and thus the inner structure of the **PhysicalObject**. When using objects from a library, these internal links are usually already present and do not have to be modeled again. In contrast, the Relations between different physical objects describe the current situation in the world, which is typically application specific and thus an important part to be defined as application geometry (cf. section 4.3).

Apart from Frames and Relations, physical objects can also be composed of further physical objects. Thus, the model handles **PhysicalObjects** according to the composite pattern [31]. In figure 4.11, the concepts around physical objects are shown. On top of the inheritance hierarchy, the concept **Entity** appears, which describes a material or immaterial item in the environment, which can be uniquely identified (cf. [39]). **PhysicalObjects** as well as **Frames** and **Relations** are entities, and can be composed within a **PhysicalObject**. Physical objects may be augmented with

<sup>4</sup>While this work concentrates on rigid bodies, extensions to deformable objects are possible as long as the expected deformation can be modeled or measured, or is irrelevant for the desired task.

further properties describing different aspects of the object. Typical properties are shape data such as maps or 3D meshes, physical properties such as mass, center of mass and moments of inertia, and further information required in an application to describe the state of or the knowledge about an object.

Typical examples of physical objects in industrial and mobile manipulators are **Links**. A Link describes a moving part of a robot arm that connects two **Joints** or the first or last Joint to its surrounding structure, and contains two Frames. The first Frame is the base of the Link, while a second one represents the position where the next linked object is connected. These two Frames are within the same rigid body and thus have a constant Transformation, modeled as a static Relation.

To connect two Links of a robot arm by a revolute Joint, a Relation can be used that is variable (the robot can move), persistent (the robot probably will not be disassembled during operation) and unknown (the exact Rotation is not known). The Relation can also constrain the Transformation to a rotation around the Z axis in a certain range, not allowing any translation. By using these constraints (that can for example map from a Joint angle to a Transformation in Cartesian space), it becomes possible to perform generic Kinematic calculations of the described mechanism, or to analyze reachability for the given robot.

When looking at the inner structure of physical objects, mainly two kinds of Relations appear:

**StaticConnections** are Relations that are constant, persistent and known. These are used to connect Frames or PhysicalObjects that are rigidly connected.

**DynamicConnections** are Relations that are variable, persistent and unknown. They appear wherever different parts of a physical object can be moved, either passively or controlled in an actuator. Different dynamic connections can for example constrain the motion to a single rotation or translation axis or to a plane, depending on the type of mechanism modeled.

The following sections describe how the reusable physical objects in the application examples are modeled.

#### 4.2.1 Mobile Manipulator – youBot Platform, Arm and Gripper

The mobile manipulator consists of the youBot platform, an arm and a gripper, which are modeled as individual physical objects.

##### youBot Platform

The youBot Platform is modeled as five rigid bodies – the main body and the four Mecanum wheels. The main body defines its base Frame and further Frames for mounting components. Figure 4.12 shows the software model of the main body and its relation to the parts of the

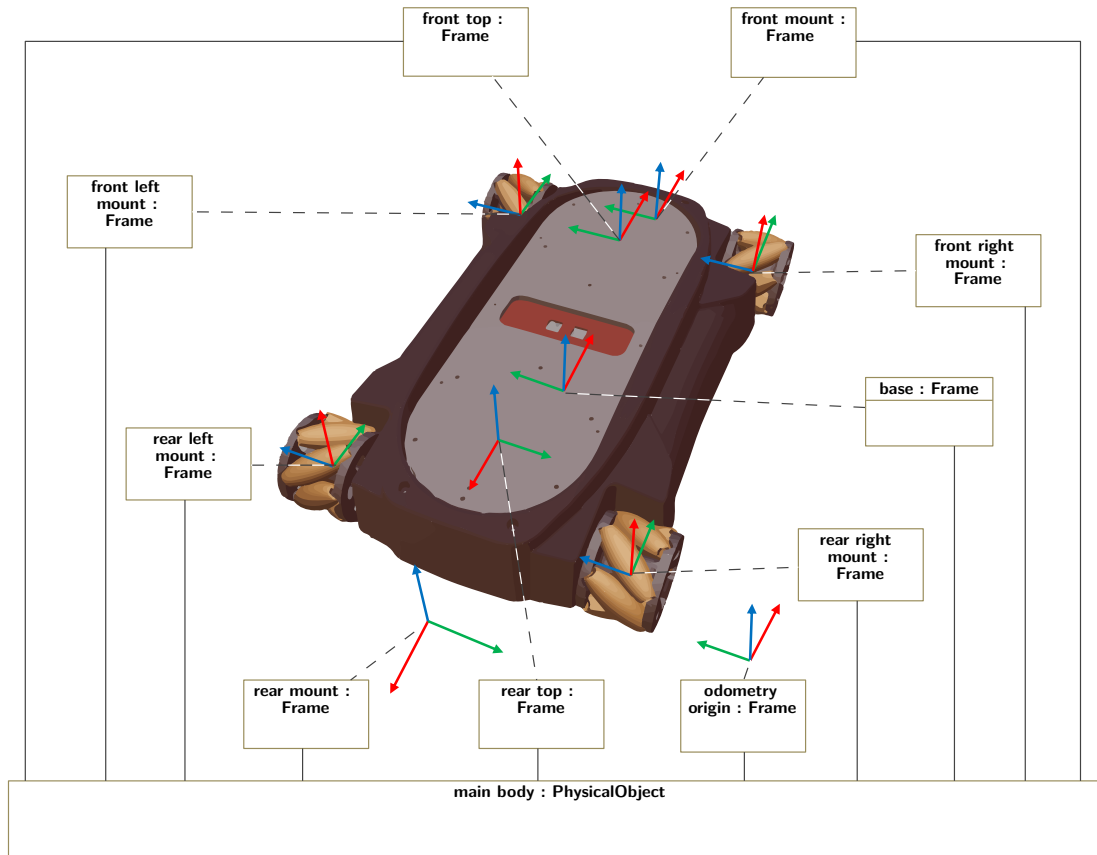


Figure 4.12: The youBot platform modeled as a physical object

youBot platform. Dashed lines link the individual Frames to the positions of the corresponding coordinate systems. As a geometric model, these Frames are connected to the base Frame through a `StaticConnection`: Four Frames where the wheels are mounted, oriented so that the Z axis points along the rotation axis, two Frames on top that can be used to mount a youBot arm or aluminum plate and two Frames in the front and back to mount sensors such as the Hokuyo laser scanner. The Mecanum wheels only contain one Frame representing the base, again with the Z axis pointing along the rotation axis. The wheels are mounted to the corresponding mount Frames through a `DynamicConnection` that constrains motion to a rotation around the Z axis.

To model that the platform can drive around, one further Frame called *odometry origin* is modeled. This Frame represents the position where the youBot platform initially started. When the wheels rotate, the Transformation between the *odometry origin* and the platform base Frame is expected to change according to the kinematics of the Mecanum wheels. This Relation is again modeled as a `DynamicConnection`, basically restricting the Transformation to a translation in the X-Y-plane and a rotation around the Z axis.

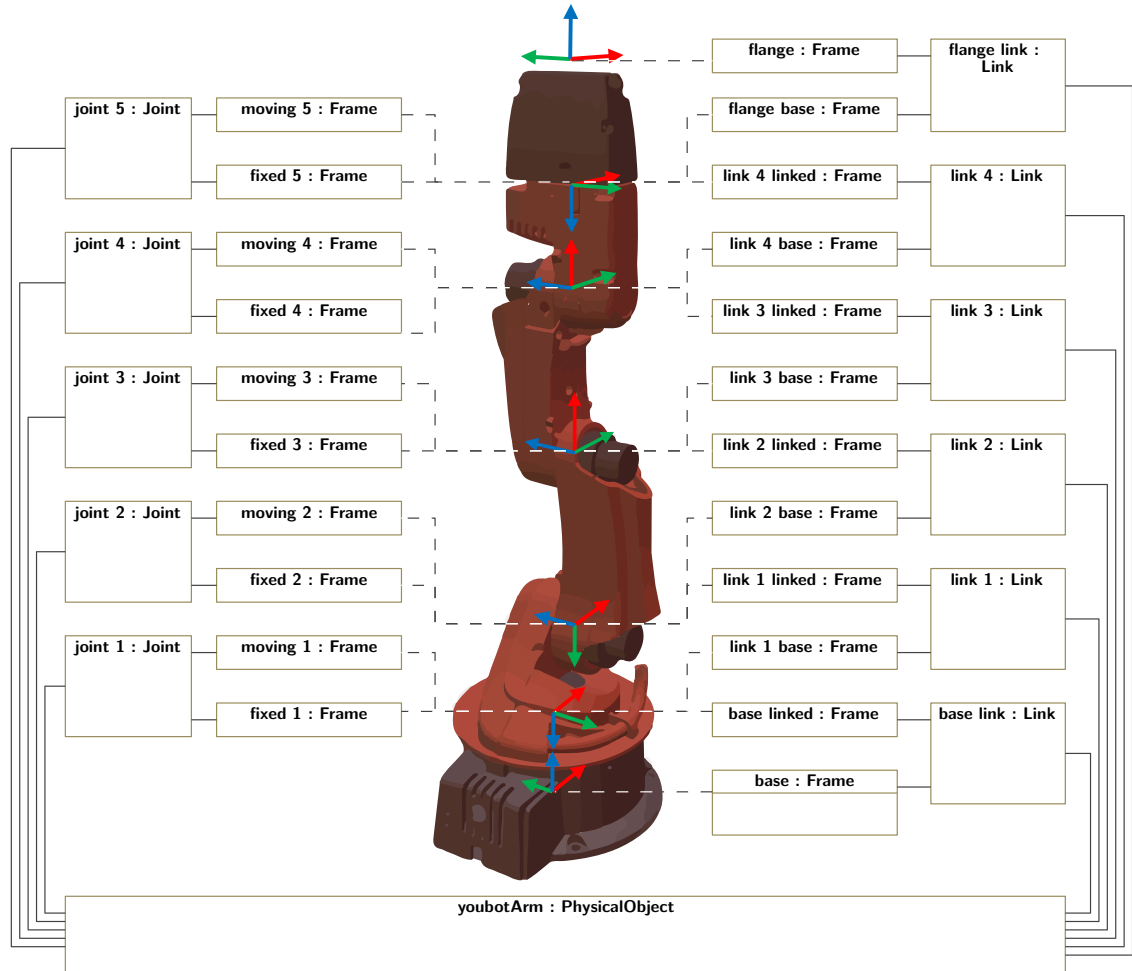
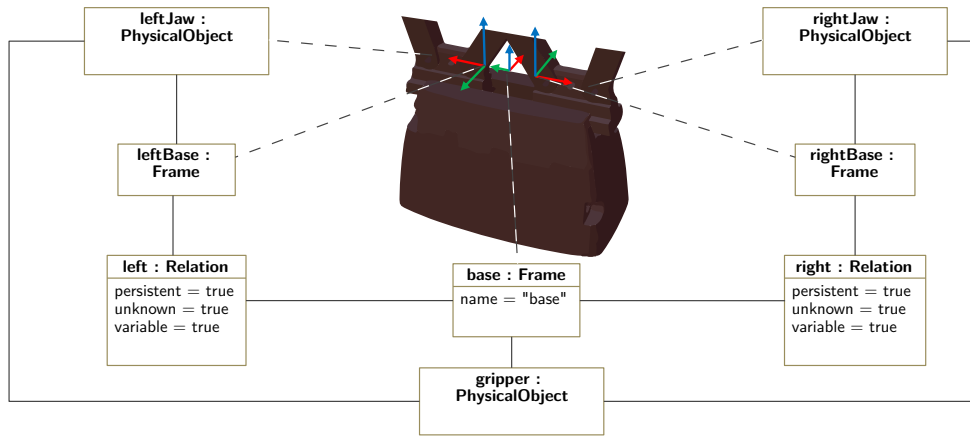


Figure 4.13: The youBot arm modeled as a physical object

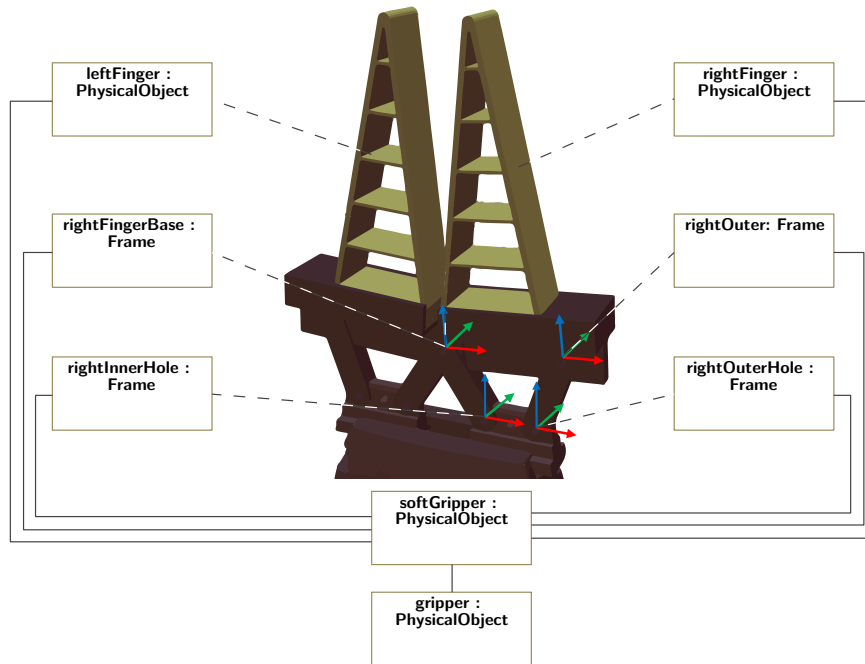
### youBot Arm

The youBot arm consists of six rigid bodies called Links, connected by the five controllable Joints (cf. figure 4.13). Each Link consists of two Frames connected by a StaticConnection. The Joints represent the motors and gearboxes present in the robot arm that allow to change the angle between their two adjacent Links. Thus, they are modeled as two Frames, called *fixed* and *moving* Frame, that are connected through a DynamicConnection to control the current angle around the Z axis.

The youBot arm has its base Frame within the first Link, and provides a *flange* Frame in the last Link that can be used to mount tools, or to specify direct motions if no special tool is used (cf. chapter 5). Between the *base* and *flange* Frame, six Links are modeled along with five Joints that allow rotation around their respective Z axis, thereby allowing to control the position of the *flange* Frame.



(a) Geometric model of the gripper and its jaws



(b) Geometric model of the soft fingers

Figure 4.14: The youBot gripper modeled as physical objects

### youBot Gripper

The youBot gripper is modeled as three rigid bodies, one fixed base and two moving jaws that can be used to mount fingers (cf. figure 4.14(a)). The jaws are connected to the base through a `DynamicConnection` limiting the motion to only allow translation in X direction.

The soft fingers chosen for the case study use a closed kinematic chain to transform the translational motion of the jaws into a rotational motion of the soft fingers (cf. Figure 4.14(b)). This mechanism is modeled as rigid bodies, using explicit computation rules to find the position

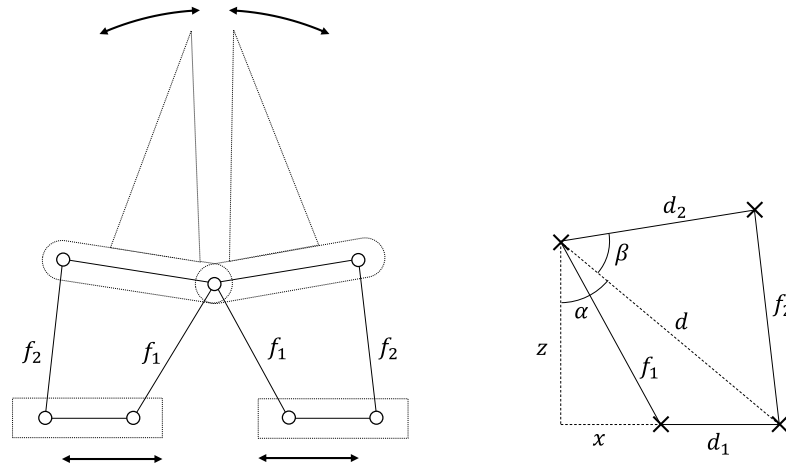


Figure 4.15: The youBot soft fingers and their closed kinematic chain

and orientation of the corresponding Frames. While it would also have been possible to model the mechanism as four unactuated Joints with two Links, the more direct approach was chosen to avoid the need for a solver supporting the closed kinematic chain. The soft fingers were also modeled as rigid bodies, because their deformation has no effect on the application programming and can thus be neglected.

In figure 4.15, the kinematic chain of the soft gripper is shown. The left shows a schema of the gripper, while on the right a model of the right half of the gripper is given, along with named distances and angles. The jaws shown in the bottom can be moved to the left and right (thereby changing the variable  $x$ ), which changes the position of the gripper fingers (especially  $z$ ) and their orientation ( $\alpha$  and  $\beta$ ). Using the Pythagorean theorem, the variables  $z$  and  $d$  can be calculated based on the jaw distance  $2 \cdot x$  and the constant link lengths  $f_1$ ,  $f_2$ ,  $d_1$  and  $d_2$  of the gripper.

$$z = \sqrt{f_1^2 - x^2}$$

$$d = \sqrt{z^2 + (x + d_1)^2}$$

Using trigonometry, the corresponding angles  $\alpha$  and  $\beta$  can be derived:

$$\alpha = \arccos\left(\frac{z}{f_1}\right)$$

$$\beta = \arccos\left(\frac{d_2^2 + d^2 - f_2^2}{2 \cdot d \cdot d_2}\right)$$

Based on these computations, the connection between gripper jaws and soft fingers are created as variable, persistent and known Relations depending on the current opening width of the gripper. More details about this implementation are given in section 6.1.1.

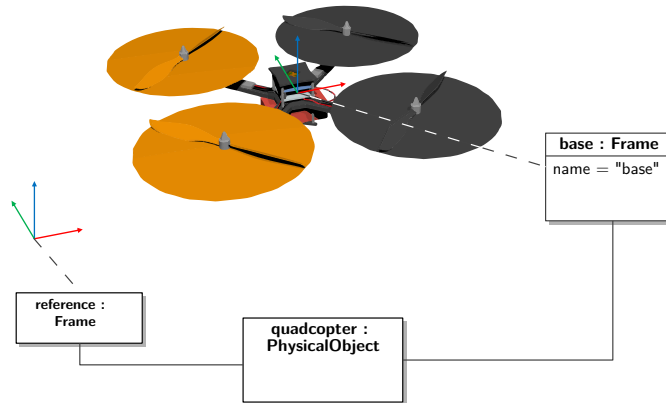


Figure 4.16: The Saphira Quadcopter modeled as a physical object

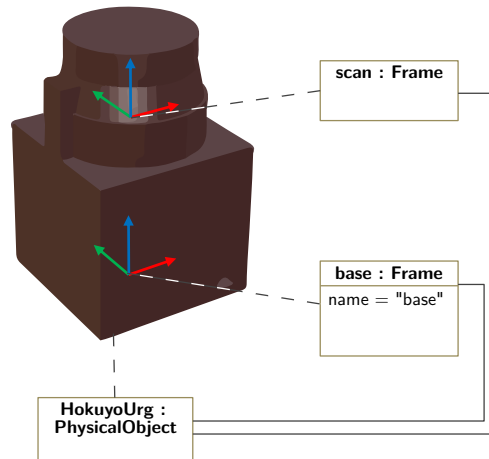


Figure 4.17: Hokuyo laser scanner modeled as a physical object

#### 4.2.2 Quadcopter – r0sewhite Saphira with Autoquad Flight Controller

The Quadcopter is modeled as a single rigid body with a *base* Frame in the bottom of the structure (cf. figure 4.16). Additionally, it defines a *reference* Frame connected through a dynamic connection that represents the start position of the quadcopter. This Relation places no constraints on the possible Transformation, but allows arbitrary translations and rotations.

#### 4.2.3 Local Sensing – Hokuyo Laser Scanner

The geometry of the Hokuyo laser scanner is modeled as a single rigid body, consisting of two Frames (cf. figure 4.17): The *base* Frame is placed at the bottom and can be used to mount the sensor on a surface or youBot sensor mount, while the *scan* Frame defines the plane of the laser scan. In the X-Y-plane of the *scan* Frame, a number of *pointCount* range measurements is performed starting at *startAngle* and ending at *endAngle*, interpreting the X direction as an angle of 0.

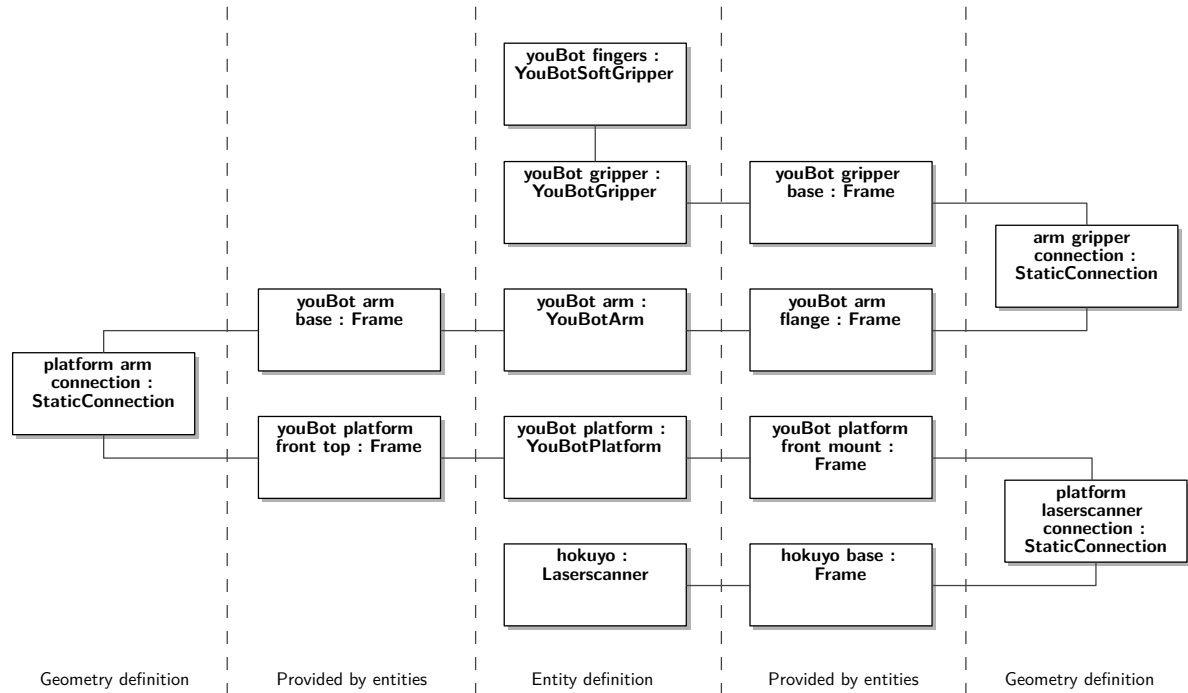


Figure 4.18: Entity and geometry definition for a youBot platform as a mobile manipulator

#### 4.2.4 Global Position Reference – Vicon Tracking System

When looking at applications, the Vicon Tracking System works as a helper to find the position of rigid bodies in the scene. However, the exact geometry of the system – its cameras and further components – usually has no direct influence on the application. Thus, the tracking system itself is modeled as a single Frame called *Vicon origin* that serves as a base for providing positions of tracked objects.

Tracked objects are modeled as **ViconSubjects** that reference their corresponding Vicon System and store the name of the tracked subject. They consist of a Frame representing the object, linked to the corresponding *Vicon origin* through a DynamicConnection without constraints.

### 4.3 Setting Up the Environment

To describe the environment an application is intended to work in, the corresponding Physical-Objects have to be instantiated and connected through Relations. In this context, three typical Relation types occur:

- A **Placement** that is constant, transient and known. It is used whenever an object is placed in a known location, but can be moved or removed.
- An **UnknownPlacement** that is constant, transient and unknown. It is used in cases when



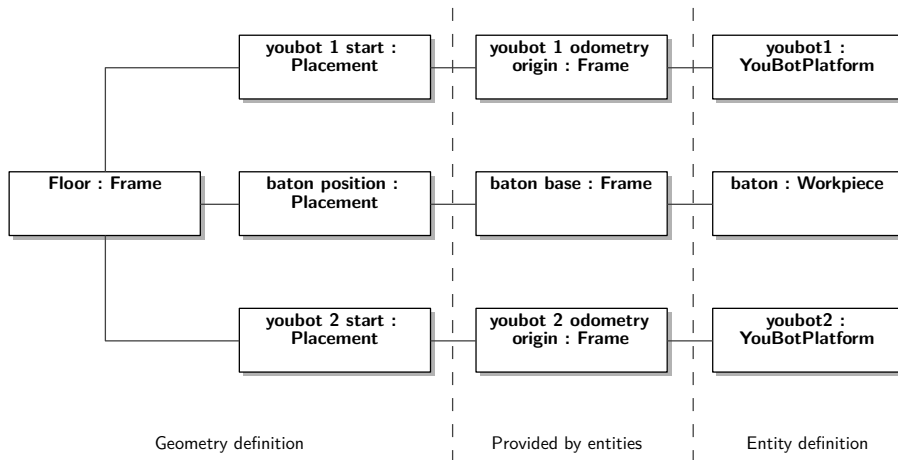


Figure 4.19: Application geometry for the handover scenario

an object is known to be in a certain place (“It is placed somewhere on the table”), but the exact position is unknown. This is especially used to describe the environment of mobile robots, where objects typically do not have a precisely known position.

- Again, StaticConnections are used to model that in the given scenario an object is permanently mounted to another object.

Using these types of Relations, an initial version of the application geometry can be constructed. At run time, Placements can be added or removed to reflect the changes performed by the actuators or other actors, while (static or dynamic) Connections remain in place.

For an application, the geometry can be defined through an XML configuration file. This configuration file describes the PhysicalObjects to be instantiated, along with their configuration parameters. Additionally, further Frames and Relations are defined, using the three aforementioned Relation types to connect the PhysicalObjects and further relevant positions as required in the application.

In the case study applications, the youBots are used as mobile manipulators consisting of platform, arm, gripper and soft fingers. For this setting, the corresponding devices have to be instantiated and linked (cf. figure 4.18). For the *youBot platform*, its *front top* Frame has to be linked to the *base* Frame of the *youBot arm*, while the *arm flange* Frame has to be linked to the *gripper base* Frame. These links are all permanent, so StaticConnections are used. The *soft fingers* have to be configured with their *gripper*, and establish the required geometric links themselves. Additionally, the laser scanner is linked to the *front mount* Frame. Using this object and relationship definition, the static structure of the youBot as a mobile manipulator is defined.

When looking at the youBot interaction example, the environment consists of two youBots and one work piece (cf. figure 4.19). The two youBots are instantiated as mobile manipulators (as described above), and are placed on the ground, thus their *odometry origin* is connected to

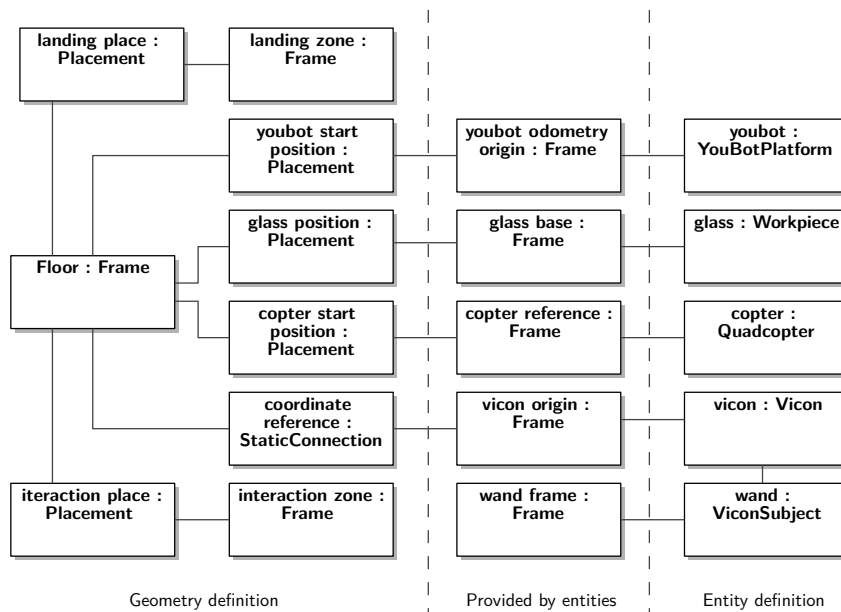


Figure 4.20: Application geometry for the gesture control scenario

the *Floor* Frame through a Placement. This set-up allows the youBots to move (because their internal DynamicConnection can change), while giving a starting position in the Transformation of the Placement. Similarly, the *base* Frame of the *baton* is connected to the *Floor* Frame through a Placement.

For the gesture control application, one youBot and one quadcopter are used (cf. figure 4.20). There the youBot *odometry origin* Frame is connected to the *Floor* Frame through a Placement, as well as the *quad reference* Frame. Additionally, the Frames representing the landing zones are placed on the floor. To access the Vicon tracking system, its origin is linked to the *Floor* Frame, and a ViconSubject for the wand (with its corresponding Frame) is used.

Using this geometry description is enough to handle the scene in a perfect or simulation case; however when uncertainty comes into play using real mobile robots, further details have to be modeled, and changes are required as described in chapter 8.

## 4.4 Comparison with Former Work on the Robotics API

The geometric model introduced in this thesis is based on the work of Angerer [2, chapter 7]. The basic data types Vector, Rotation, Transformation and Twist along with their operations were adopted unmodified, while extensions were introduced for the data types with metadata. In the case of Points, it is now possible to specify an Orientation to interpret the Vector in, simplifying the specification of certain Points and providing consistency with the Twist data types.

Additionally, Pose and FramePose were added as a semantically enriched version of Transformations which can easily be converted into other reference Frames or Orientations, thereby

fulfilling design goal 4.3. Previously, these calculations were only available within *converters* [89] when transforming Commands to RPI primitive nets and could not be used independently in applications, while position specifications had to use Frames or Transformations with implicitly given semantics. Now, Pose and FramePose fill the important gap between lightweight Transformations without semantics and heavyweight Frames with identity, simplifying programming and avoiding errors caused by incorrect composition of the geometric primitives (cf. [22]).

Finally, Velocity and FrameVelocity were added to complement the *VelocitySensor* introduced by Angerer [2], so that each geometric sensor can provide a data type that includes all semantic information of the sensor, and velocities can be specified consistently. For each of the geometric data types, a *Sensor* type (now called *RealtimeValue*, cf. section 6.1.1) is provided, similar to the *Sensors* introduced by Angerer [2], but with the same extensions concerning semantics and methods to change the metadata using dynamic information from the world model. These concepts are important for mobile robotics, where managing velocities is vital for estimation and prediction, while lightweight position descriptions allow to exchange geometry information between cooperating robots.

Looking at the *Frame graph* consisting of Frames and Relations, the handling of Relation types has been changed: While in the work of Angerer [2], the characterization between Placements, DynamicConnections and StaticConnections was performed through inheritance, the different facets are now described through independent properties. This change for example allows Placements describing less structured environments to have a variable Transformation, such as when an object placed on a table is observed through a sensor that provides the Placement with up to date position information. Additionally, unknown Relations were introduced to cope with the challenges of environment modeling present in mobile robotics.

Working with different views of the world, Angerer [2] suggested providing exactly two sets of values, one for commanded and one for measured positions. This choice is now extended to the more generic concept of *FrameTopologys*, allowing to work with views of different runtime environments and distributed robots, fulfilling design goal 4.4.

Looking at the description of devices and physical objects, the chosen solution is similar to Angerer [2] and Hoffmann [39]. However, Devices no longer reference and require a *DeviceDriver* to set up their geometry, but can instead use unknown Relations to describe aspects that can change during run time. This way, it is possible to completely configure the application geometry without having to specify deployment issues (cf. design goal 4.5), and to work with multiple runtime environments (cf. section 9.1) as required when controlling multiple distributed robots.

## 4.5 Related Work

Looking at other robot frameworks, the use of vectors, matrices and quaternions is widely agreed on. However, when it comes to modeling and expressing the semantics of geometric

constructs, differences occur:

In ROS [82], the basic geometric concepts are expressed through data types defined in the *geometry\_msgs* package. There, rotations are expressed as *Quaternion*, while *Point* and *Vector* describe positions and displacements. *Pose* and *Twist* are used for complete 3D poses and velocities, all using *float64* as a data type for the values. For all these types, a *Stamped* variant exists (e.g. *PointStamped*, *QuaternionStamped* and *PoseStamped*), which additionally includes a header with a sequence ID, a time stamp and the name of the reference frame. To define the frame graph of an application, geometric transformations of frames relative to each other are published at variable frequency. This way, each frame can be assigned a parent, relative to which the transformation is published. Transformations are either published by *device driver nodes* when new dynamic information becomes available, or by *static transform publishers* for constant transformations. Using the *tf* library (maintained by Foote [29]), it is possible to compute transformations between different frames, and to transform a stamped geometry data type into another reference frame, using the frame topology defined by the published transformations. Additionally, it is possible to transform the data using transformations from the past, in order to correctly handle older data. All these data types share the commonality that the orientation to interpret the data in cannot be given independently, but is assumed to be the reference frame. Additionally, for twists no pivot point can be given, so that some motions cannot be specified in a straightforward fashion. Furthermore, the transformations published for *tf* do not carry semantic information, making it hard to distinguish between constant, variable or persistent relations based solely on the frame graph.

In the OROCOS project [14], the KDL library [95] provides basic support for geometric data types, along with operations to perform basic computations (such as composition of transformations and the change of pivot point for rotations). In this context, De Laet et al. [22] introduce an extensive description of the semantics of geometric primitives. These semantics are similar to the metadata introduced in sections 4.1.2 to 4.1.5, however with two differences: De Laet introduces rigid bodies as first-level citizens, and bases all geometry descriptions on them. There, a *Pose* describes the position and orientation of a frame on a rigid body relative to another frame on another rigid body, and an *Angular Velocity* only talks about two rigid bodies, without giving any frames. Similarly, *Twists* describe the motion of a rigid body relative to another rigid body, by giving the angular velocity of the bodies, together with the linear velocity of a chosen point. This chosen point has semantics similar to the Pivot point, however this formulation does not easily allow to describe the motion of a given frame on the moving body. When coordinates are given for a geometric data type, special *Coord* variants are used (such as *PoseCoord* and *TwistCoord*), which include an additional coordinate frame that is used as an orientation (to describe the meaning of the X, Y and Z axis). In summary, this representation provides similar flexibility and expressiveness, however with more focus on rigid bodies. Solutions by De Laet et al. [21] and Blumenthal et al. [9] allow to set up scene graphs

for an application, and to find or convert transformations between different frames. However, in a context focusing on Frames such as the one given in this thesis, a representation that describes geometric properties of Frames is more flexible: It can handle Frames that do not belong to a rigid body, and directly talks about the behavior of given Frames. Talking about Frames and their velocity becomes especially important when defining motions for robots, where an exactly chosen frame should perform a motion and reach the specified goal – it matters whether the tip of one gripper finger or the center of the gripper arrives at the given position.

As opposed to the suggested *Frame graph*, many *Scene graph* implementations use a different choice of nodes and edges: While the *Frame graph* uses Frames as nodes and Relations as edges, *Scene graphs* [9, 106] often model both Frames and Transformations (and further concepts such as sensor data) as nodes that are connected by edges without further semantics. While providing a bit more flexibility, this approach requires to define a well-founded meaning of Transformation nodes or Frame nodes linked directly without an instance of the other type of node in between, often mixing concerns by defining Frame nodes to have an identity transformation.

In the ROS ecosystem, apart from manually publishing all elements of the robot structure, URDF [66] is often used to set up robot structure and kinematics. Therefore, links and joints are defined in an XML format, together with geometry, physical and collision properties. Based on these robot description files, a robot model with its kinematic structure can be loaded into a ROS environment. Similarly, COLLADA [6] can be used to describe the geometric model and link structure of a robot.

However, both methods only yield generic objects for the robot parts. In contrast, the object-oriented approach suggested in this thesis models the parts as detailed objects, allowing to include special properties or features to the created objects as suggested by object-oriented design [87]. Similarly, without detailed object-oriented modeling, special behavior such as the motion of the soft gripper is limited to software frameworks that can automatically solve the closed kinematic chain. Thus, the approach chosen in this thesis can be used in a more flexible way, while it is still possible to implement a COLLADA or URDF parser to integrate existing models from other frameworks.



## MOTIONS AND TOOL ACTIONS

When a static view of the environment in the application geometry has been modeled (cf. chapter 4), the desired behavior can be described. Therefore, tasks such as motions and tool actions are defined and assigned to active objects for execution. According to Requirement 2, the specification of motions should be extensible, and possible in Cartesian and Joint space. When working with stationary robot arms and mobile systems, it is desirable to allow using the same motion specifications for all these types of robots.

**Design Goal 5.1.** *Consistently define motions for stationary, mobile and flying robots – using the same type of specification for the same type of motion on different devices.*

Additionally, different aspects of the motions and their execution should be configurable. This includes aspects of the motion itself (e.g. the motion profile of the trajectory), but also aspects that affect how the motion is executed by the Actuator.

**Design Goal 5.2.** *Parameterize motions and their execution – with common parameters for the motion and device-specific parameters for the execution on concrete devices.*

Looking at motions in Cartesian space, the kinematic properties of devices play an important part. While most classical industrial robots have six degrees of freedom and can thus reach every position in every orientation, this is no longer true for mobile robots (and mobile manipulators). While mobile robots are usually limited to motions on the ground and thus three degrees of freedom, for mobile manipulators the total number of degrees of freedom typically exceeds six, however split into the locomotion system and the manipulator(s). So, some devices do not have enough controllable joints to reach the six degrees of freedom of Cartesian space (e.g. the youBot arm with five joints), while others are redundant and offer more degrees of freedom than required (e.g. the youBot platform together with the arm). Both cases may occur when

executing Cartesian motions on mobile robots, requiring to specify a strategy how to work with these kinematic properties.

**Design Goal 5.3.** *Handle kinematic restrictions or redundancy in specified motions – as configured by the corresponding application.*

To achieve these design goals, a solution based on the work of Angerer [2] has been developed. In this solution, tasks are modeled as an extensible set of Actions. In this context, **Actions** are the smallest entities of task descriptions that describe a desired state (or sequence of states) of an Actuator, independent from the concrete Actuator instance that will execute them. Section 5.1 describes the different Actions used to specify motions and tool operations.

Active objects that can provide data or are able to execute Actions are called Devices and are modeled as specialized PhysicalObjects. **Devices** represent any kinds of mechatronic devices that can be accessed, while their specialization **Actuators** represent devices that are controllable and can influence their environment. To adapt these Actions to the capabilities of a concrete Actuator and to control the exact behavior of the Actuators, configuration parameters can be given as DeviceParameters, as detailed in section 5.2.

Section 5.3 gives an overview about the changes that were applied to the work of Angerer [2], while section 5.4 compares the solution to other approaches to define motions and actions.

## 5.1 Actions to Describe Motions and Tool Operations

Actions serve as a device-independent specification of a task to be executed. For example, an Action describes an entire trajectory in Cartesian space, or a position where a robot should move to. From a control point of view, Actions provide set-points as an input to a controller for the Actuator requested to execute the Action. Looking at Actions, two fundamentally different types can be found:

**Goal Actions** describe a goal that the Actuator should eventually reach. It is within the responsibility of the Actuator to find and apply appropriate system inputs that will eventually lead to the given goal. Thus, the Action gives no exact time *when* the goal is to be reached, however the fastest possible solution respecting the configured parameters is preferred.

**Process Actions** specify continuous processes and describe a set-point sequence or trajectory the Actuator should exactly follow. Each set-point is expected to be reached immediately, before a new set-point is provided in the next interpolation cycle. From a control point of view, executing Process Actions demands a good tracking performance of an Actuator, while the Action has to make sure the given set-points are feasible (i.e. reachable in the next time instant).



An Action is called **Completed** whenever it reports a set-point for which an Actuator that has arrived at the set-point fulfills the purpose of the Action. For a trajectory as a typical Process Action, this happens once the end has been reached (and provided as a set-point), while an Action specifying a goal for a robot arm to move to (as a typical Goal Action) is completed immediately. However, for other Goal Actions completion can also occur later, especially when working with moving goals as described in the following sections.

During execution, an Action can be completed while the corresponding Actuator has not yet reached the given set-point. This is especially common with Goal Actions that report completion immediately, while the goal can be far away from the current Actuator state. In this case, execution continues until both Action and Actuator are Completed, meaning that the Action-provided set-point fulfills the purpose of the Action and the Actuator has reached the set-point.

Looking at Actuators, different types of Actuators motivate different types of Actions: While some Actuators naturally work in Cartesian space, others (due to redundancy or kinematic constraints) can better be described in configuration space (talking about the independent degrees of freedom).

For a manipulator with  $n$  revolute Joints, the configuration space  $C \subset \mathbb{R}^n$  describes the allowed range for each of the Joints and is given as the cross product of the corresponding intervals of angles. Each configuration in  $C$  can be mapped to Cartesian space (into a position and orientation of the end effector) using the forward kinematics function, however this mapping does not have to be injective or surjective: Multiple configurations can point to the same pose (e.g. due to singularities or redundancy), and some poses cannot be reached (if they are outside the working envelope). So, specifying motions in configuration space is more powerful, however within the reachable region Cartesian space motions can also be used as a more intuitive representation.

For an electric parallel gripper, the configuration space is one-dimensional describing the interval of allowed opening widths. The current configuration of a gripper can be mapped to the Cartesian position of the fingers, allowing to specify a desired finger distance (taking into account the geometry of the fingers).

Apart from continuous intervals or subspaces, discrete states can occur as a (parts of) configuration space. Pneumatic grippers for example can only be open or closed, so the configuration space is binary. Other devices do not influence Cartesian positions, such as controllable lights or speakers, or digital outputs used to switch power on or off. These have their configuration space, but no correspondence to Cartesian space, and thus can only be controlled in configuration space. Similarly, operation modes or parameter configurations of Actuators can form a discrete part of the configuration space and can be handled by specific Actions.

Section 5.1.1 takes a look at motions in Cartesian space, followed by motions in configuration space (Section 5.1.2) and tool operations (Section 5.1.3).

Type	Completion	Goal	Process
Position	immediately	MoveToPose	
	fixed time		CartesianTrajectory
	when canceled	FollowPose	HoldPose
Velocity	when canceled	FollowVelocity	HoldVelocity
		CartesianJogging	

Figure 5.1: Cartesian space motions

### 5.1.1 Motions in Cartesian Space

Motions in Cartesian space can be separated into goal and process Actions. Additionally, they can be differentiated by the type of set-point they provide (position or velocity), and by their duration (if they run until canceled, have a fixed duration or complete immediately). Most Cartesian space Actions can be found in the field spanned by those three dimensions as shown in figure 5.1.

Figure 5.2 gives an overview of the software model of different types of Cartesian motions, along with the parameters required to specify their main properties. In position space, **MoveToPose** and **FollowPose** work as goal actions. They share the characteristic that they provide a Pose as a goal to an Actuator. However, the difference is that MoveToPose immediately signals completion and thus execution ends once the Pose has been reached, whereas FollowPose only reports completion when canceled, and thus continues to follow the Pose even after the goal has been reached once.

**CartesianTrajectory** as a Process Action describes an exact trajectory in Cartesian space and is typically used for manipulators, but also occurs for mobile robots as long as the described path can be executed by the robot according to its kinematic constraints. Typical trajectories include straight line motions (**LIN**), circular motions (**CIRC**) or curved motions specified by a cubic Bézier curve (**SPLINE**). They provide a time-dependent Pose to the Actuator that is to be followed (i.e. reached instantly), and complete once the final Pose of the path has been reached.

These CartesianTrajectorys share the characteristic that they start and end in a stable state (i.e. the velocity is zero). Furthermore, these motions may only be specified between Poses with reference Frames that are statically connected and do not change at runtime, as they describe an exactly defined and precomputed path. An example about rectilinear motions can illustrate the reason for this limitation: Assuming that a robot is requested to go to a given point in a straight line, however in the middle of the motion the point moves to the side, the resulting motion can no longer be rectilinear. Thus, these kinds of motions are forbidden for CartesianTrajectorys. However, two alternatives can be used: The first way is to find the current Transformation of the goal Pose relative to the start Pose and use it to define a new, non-moving Pose. This way, the robot can execute the motion, however it will not arrive at the position where the specified Pose is at the time of arrival, but at the place where this Pose was when the motion was planned.

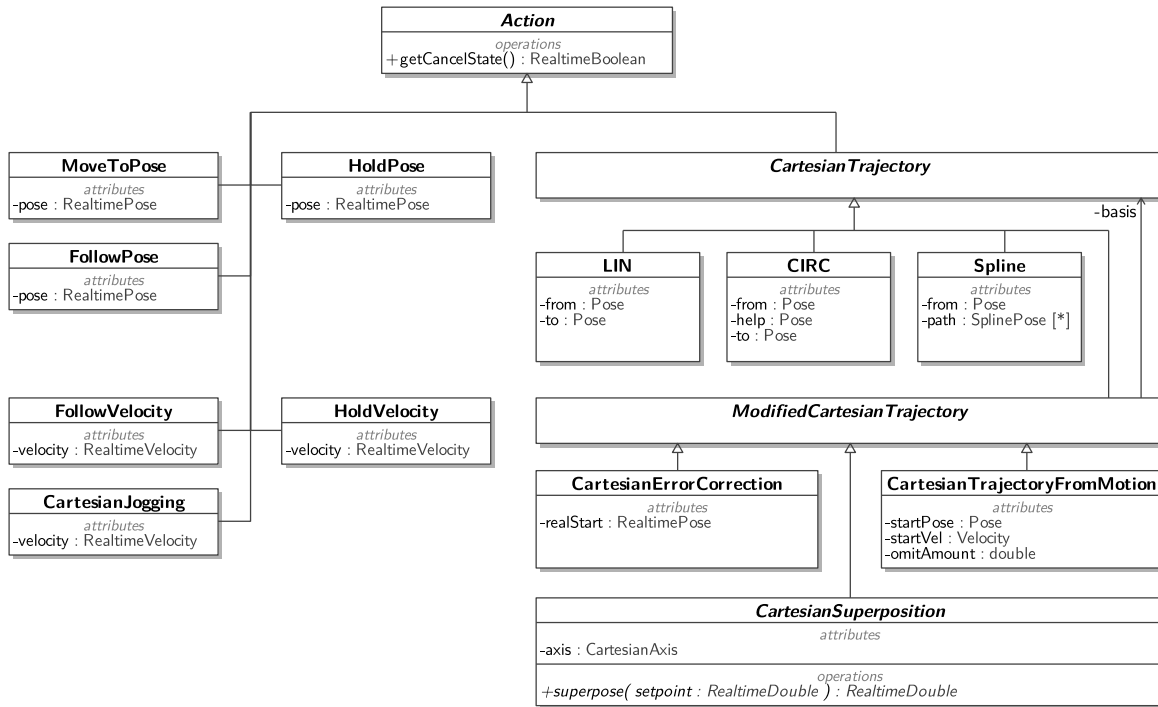


Figure 5.2: Class diagram of available Cartesian space motions

This is enough if the goal does not move. If the goal moves, goal motions can be used (e.g. **MoveToPose**) that promise to reach the given goal, however provide no guarantees about the exact path that will be taken.

To start a **CartesianTrajectory** from an initial situation where the robot is in motion (e.g. for motion blending, cf. section 7.2.1), a **CartesianTrajectoryFromMotion** can be used that replaces the first part of another **CartesianTrajectory** (called *basis*) by a smooth curved motion starting at a given initial position and velocity. It wraps its *basis* motion and is parameterized with the expected initial Pose and Velocity as well as a parameter controlling how much of the *basis* motion may be omitted to reach a fluent motion. It then replaces the initial part of its base trajectory by a cubic Bézier curve that starts with the given initial condition and ends at the position and velocity required to continue the given trajectory. If the initial Cartesian position of a motion is not exactly known, the **CartesianErrorCorrection** Action configured with a sensor reporting the current Pose corrects its *basis* **CartesianTrajectory** so that the difference between trajectory start Pose and the actual Pose is smoothly corrected within the first half of the trajectory. **HoldPose** is used to keep the Actuator at a given position, for example at the goal position of the previous **CartesianTrajectory**. Therefore, it provides the given Pose as a position set-point and only completes when canceled. This way, even if the previous goal moves, the robot will follow as long as the movement can be compensated instantly. **CartesianSuperposition** can be used to modify another **CartesianTrajectory**. It allows to apply mathematical calculations to

Type	Completion	Goal	Process
Position	immediately	MoveToJointGoal	
	fixed time		JointTrajectory
	when canceled	FollowJointGoal	HoldJointPosition
Velocity	when canceled	FollowJointVelocity	HoldJointVelocity
		JointJogging	

Figure 5.3: Joint space motions

one spatial dimension of the set-point provided by its base trajectory yielding the new set-point.

When proceeding to velocity space, two typical goal motions are **FollowVelocity** and **CartesianJogging**. Both provide a Velocity to the Actuator as a goal that is to be smoothly achieved by an Actuator, and thus does not need to exhibit the continuity required to be instantaneously reached by a physical device. The provided Velocity can in both cases be changed at any time. For both, the reference Frame as well as Orientation and Pivot point can be specified, allowing to specify the Velocity in a convenient representation. FollowVelocity is used for application-defined motions, e.g. specifying a driving direction of a mobile robot, and only completes when canceled. CartesianJogging is used to manually control a robot, e.g. by a user through a gamepad or graphical user interface. It only completes when canceled, to allow the user to control the speed as long as required.

For smooth, computer-generated velocities, **HoldVelocity** can be used that works as a process Action and provides a time-dependent Velocity to be immediately achieved by the Actuator. It as well has no predefined end and thus completes once canceled.

Additionally, there are Actions that for a given Actuator convert a configuration space motion into Cartesian space (using the forward kinematics function), to allow modifying it with Cartesian corrections.

### 5.1.2 Motions in Joint Space

Motions in configuration space talking about one or multiple robot Joints can again be differentiated in three dimensions: They can describe a path or a goal, can talk about positions, velocities or accelerations, and can end immediately, after a fixed time or run until canceled.

Figure 5.3 gives an overview about different motions in configuration space, while figure 5.4 goes more into detail with parameters required to specify the corresponding behavior. Looking at position space, two goal Actions are available: **MoveToJointGoal** commands an Actuator to move to the given joint position, and immediately reports completion (so that the motion ends once the Actuator reaches the given position). **FollowJointGoal** commands the same, however follows the goal if it is changing, and completes when canceled. Process Actions in configuration space include all kinds of planned **JointTrajectories**, such as Point-to-Point-Motions (**PTP**) or Bézier curves in Joint space (**JointSpline**) that work between fixed configurations. **HoldJoint-**

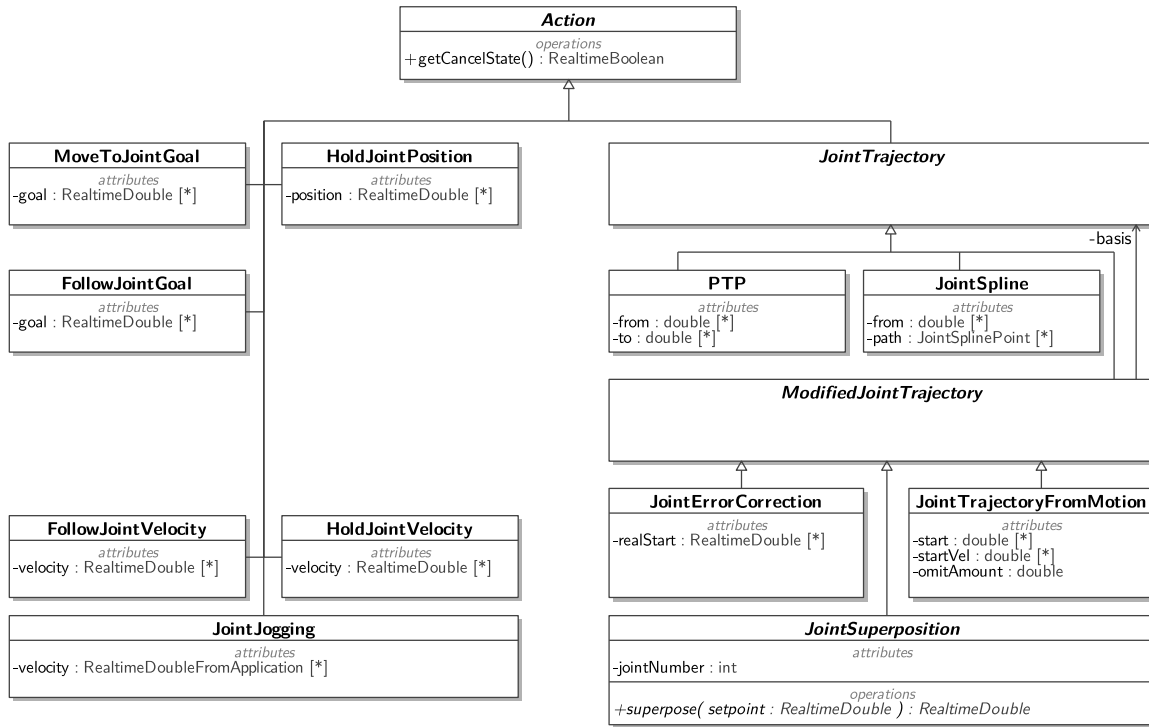


Figure 5.4: Class diagram of available Joint space motions

**Position** is used to guarantee that a (maybe time variable) configuration is maintained, while **JointSuperposition** allows to modify another **JointTrajectory** by applying mathematic calculations to some of the joint values. Similar to the Cartesian case, **JointTrajectoryFromMotion** and **JointErrorCorrection** allow to work with moving or imprecise start positions.

In velocity space, **FollowJointVelocity** and **HoldJointVelocity** are available, specifying a joint speed to apply, with the difference that the speed given in **HoldJointVelocity** must be smooth and reachable and will be applied at once, while **FollowJointVelocity** allows jumps in the input that will be followed as fast as possible. These Actions are especially helpful for unlimited joints, e.g. a motor that drives a conveyor belt, where talking about the velocity is more natural than talking about the position. Additionally, **JointJogging** is provided to manually control the positions of the robot Joints from an application, e.g. when manually teaching positions of a youBot arm. As a Goal Action in configuration space, it accepts unfiltered inputs that do not have to respect the acceleration or jerk limits of the robot.

### 5.1.3 Tool Operations and State Transitions

Looking at the youBot gripper, only goal motions in position space are supported that define where the fingers should move to. However, the gripper does not allow to interrupt or update the goal during motion. In this special case, the **MoveGripper** Action is used to move the gripper

to a given finger distance. It can be used to execute open and close behavior, as well as for cases where intermediate positions are required.

Depending on the type of fingers attached, the correct value to open or close the gripper can vary: Using the standard fingers that move parallel along with the jaws, the maximum opening width of the jaws leads to maximally distant fingers. With the soft gripper, in contrast, the finger tips move on a circular trajectory when moving the jaws, with a minimum distance when the jaws are at the maximum opening width. Correspondingly, to open the gripper correctly, the fingers have to be taken into account. Chapter 7 describes a further software layer that allows to abstract from this change.

Further Action that do not support continuous set-points can be found when working with individual operation states: For example, the youBot arm supports two different controller modes, position control and joint impedance control (as further described in section 5.2.2). The Actions **SwitchToPositionControl** and **SwitchToJointImpedance** allow to switch between these controller modes. Furthermore, a quadcopter can be enabled (“armed”) or disabled (“disarmed”), modeled through the **Arm** and **Disarm** Actions.

## 5.2 Device Parameters to Configure Motions

The Actions described in the previous sections define central aspects of the task to execute. However, to fully specify the task execution on a given Actuator, further configuration parameters are required. These parameters are modeled as **DeviceParameters**. These DeviceParameters can be given by the Actuator itself or manually along with the Action specification, and can influence both set-point generation inside the Action and set-point interpretation by the Actuator.

Typical DeviceParameters are the **JointLimitParameter** and **CartesianLimitParameter** used to constrain the possible motion of the Actuator, as described in section 5.2.1. Looking at Actuators that support multiple control modes, DeviceParameters are available to configure this aspect, as explained in section 5.2.2. For Cartesian space motions, the **MotionCenterParameter** allows to define which part of the Actuator should follow the given motion. This parameter is further detailed in section 5.2.3. However, for many Actuators with limited degrees of freedom not all Cartesian positions are reachable. To define helpful behavior in these cases, **FrameProjectors** are introduced as a new concept, projecting unreachable position to other reachable ones. Section 5.2.4 explains their concept and their use with youBots.

### 5.2.1 Cartesian Limit and Joint Limit – Limiting Motion Path and Speed

As physical objects with mass and inertia, Actuators exhibit limited dynamics. These limits, namely maximum velocity, acceleration and jerk are handled through DeviceParameters. For Cartesian space, the **CartesianLimitParameter** allows to configure the desired maximum velocity, acceleration and jerk of motions, while the **JointLimitParameter** gives the same constraints

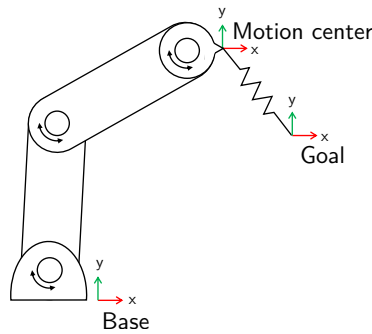


Figure 5.5: Impedance control in Cartesian space

in configuration space. Additionally, the `JointLimitParameter` specifies limits for feasible Joint positions, namely upper and lower bounds for each Joint that may not be exceeded by motions.

By default, Actuators provide `DeviceParameters` with the maximum positions, velocities and accelerations reachable by the hardware. However, for individual motions more conservative limits can be used to reduce the velocity of the Actuator or to limit the strain put onto objects handled. For these given `DeviceParameters`, the Actuator first checks if they do not exceed its known maximum parameters, before they are provided to the Action. Actions – especially Process Actions – use these parameters for motion planning purposes, to plan trajectory segments that conform to the given velocity, acceleration and jerk limits. Additionally, these limits will be respected by the Actuator when executing goal Actions. This way, the Actions themselves as task definitions remain independent from the Actuator instance that will execute them, while the resulting trajectories are adapted to make use of the full Actuator capabilities.

### 5.2.2 Controller Parameter – Controller Choice for Motion Execution

Traditional industrial manipulators aim for precision, and thus choose position controllers that provide high stiffness. However, using stiff controllers leads to high forces applied when the robot comes into physical contact with its rigid environment. To avoid this, impedance control has been suggested [41]. The idea behind impedance control is to model the robot as a mass-spring-damper system in configuration or Cartesian space.

In configuration space, each Joint is modeled as a mass that is connected to the desired goal through a spring and a damper. When the commanded position changes, the spring is stretched and applies a torque towards the desired position, proportional to the amount of displacement and the configured spring constant. Using an undamped spring however would lead to infinite oscillation, so a configurable damper is added to stabilize the system. A similar approach is used in Cartesian space (cf. figure 5.5), where the virtual spring connects the end effector (or configured motion center) to the desired pose, forcing the manipulator towards its goal.

Using an impedance controller, the force exerted on the environment when approaching a position within a rigid body is limited, thus reducing the possible damage. However, impedance

control reduces position accuracy, because position errors caused by friction are only corrected after the spring torque exceeds the friction. Additionally, carrying a load changes the reached position of the manipulator, because the manipulator will end in a position where the gravity force introduced by the load just compensates for the force exerted by the virtual spring. This way, a steady state error is introduced, unless the load is known and compensated for.

The youBot arm supports two controller modes, **PositionController** and **JointImpedanceController**, that can be selected through `SwitchToPositionControl` and `SwitchToJointImpedance`. The `PositionController` has no further settings, while for the `JointImpedanceController`, various parameters can be set for each Joint through **JointImpedanceParameters**: The *stiffness* is given in [Nm/rad], while *damping* is a relative value between 0 (no damping) and 1 (full damping). Furthermore, an *additionalTorque* can be given that will be applied to the Joint, leading to a displacement of the spring or a force applied to an object in contact. To limit the allowed torque, a *maximumTorque* can be defined, limiting the maximum torque applied due to *stiffness* and *additionalTorque*.

Through the impedance controller, it becomes possible to compensate for position uncertainties, because the compliance of the manipulator can correct minor position errors. Additionally, impedance control makes it possible to grasp an object using two manipulators without knowing their exact displacement, as long as the forces caused by the springs remain sufficiently small.

For the quadcopter, the position controller can be parameterized from the application through the **QuadcopterPositionControllerParameter**, allowing to specify gains for the position and velocity control loops. Furthermore, the **QuadcopterAttitudeControllerParameter** allows to specify the amount of thrust required for the quadcopter to hover, as well as the maximum pitch and roll angle and the parameters for the thrust integrator used to compensate for additional load on the quadcopter. For more details on the control loops of the quadcopter implementation, see section 9.2.2.

### 5.2.3 Motion Center – Specifying what Moves for more Flexible Motions

All Cartesian space Actions share the characteristic that they only describe a Pose, path or speed in space, but do not give details about which part of an Actuator is expected to reach or execute it. This information is added through a DeviceParameter called `MotionCenterParameter`. The motion center parameter specifies the Frame that is expected to reach the given point. This way, not only the flange of a manipulator or the center of a mobile platform can move along a predefined path, but also any point or object that has been mounted to the robot can be selected as a motion center.

Figure 5.6 shows an example of a rectilinear motion for a youBot platform. The dotted line denotes the desired trajectory specified in an Action, defined relative to the *Ref* frame. *X* represents the Pose at a given time instant *t*. The Frame *Ref* is statically linked to the *Origin* Frame of the youBot. The youBot as a mobile platform is able to control the connection



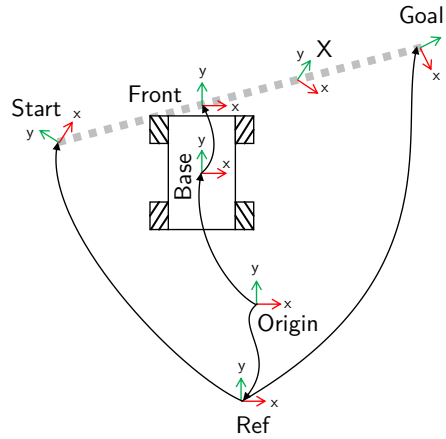


Figure 5.6: World model for a youBot platform executing a linear motion

between *Origin* and *Base* representing the moving part of the youBot. In addition to the motion specification given by the Action, the *Front* Frame has been chosen as a motion center. It is statically linked to the *Base* Frame, and is requested to move along the given trajectory. In this example, the Pose describes the desired position and orientation of the *Front* Frame relative to *Ref*. To execute the motion, the Actuator needs to know the required Transformation of the *Base* Frame relative to *Origin* so that the *Front* Frame is at the given Pose. Thus, the Poses and Velocitys provided by the Action have to be converted using the World model, as explained in section 4.1.4.

Furthermore, the position of the reference Frame of the motion does not have to be constant relative to the executing Actuator: It is perfectly valid to specify one robot's motion relative to any second robot's moving Frame. Doing this, the other robot's motion is automatically compensated to achieve the desired Pose or Velocity. For example, commanding one youBot to execute a HoldPose Action with a position that is defined as 1 m right of another youBot will make the first youBot follow the second wherever it moves to – given that the initial positions of both youBots fulfill the precondition for HoldPose.

Additionally, it is possible to use a Frame as a motion center that is not connected to the moving part, but to the fixed part of the robot, if the specified goal position is defined relative to the moving part. This way, it is easy to program a youBot platform so that a tool fixed in the world executes the desired motions on top of the robot.

#### 5.2.4 Frame Projectors – Working with Kinematic Restrictions and Redundancy

Looking at the youBot arm and platform, both share the characteristic that they cannot reach every position in 3D space within their working envelope with every orientation. Given a flat surface, the youBot platform is constrained to a position in the X-Y ground plane, while the orientation is limited to a rotation around the Z axis. The manipulator can reach all points in

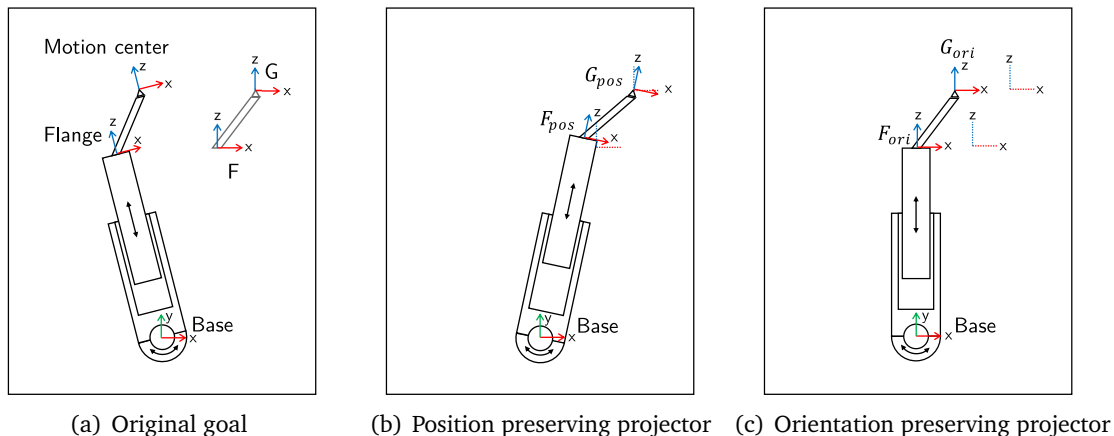


Figure 5.7: 2D manipulator with one revolute and one prismatic joint, with position and orientation preserving projector

3D space in its working envelope, however not all can be reached with every orientation.

Thus, most trajectories specified in Cartesian space cannot be exactly executed using these Actuators. To solve this problem, FrameProjectors are introduced that convert a Pose given by a motion into a Pose that is reachable by the Actuator. These FrameProjectors can be chosen through the **FrameProjectorParameter**.

To explain this concept, the example of a two-joint manipulator in 2D space is analyzed. The manipulator consists of one revolute Joint, followed by a prismatic Joint and an end effector mounted at the flange, which is defined as the motion center (cf. figure 5.7). Given this kinematic structure, the robot can reach every position in 2D space, however most of them only in one orientation.

We look at the case when it should reach the given position and orientation  $G$  with its end effector (cf. figure 5.7(a)). To reach this Pose, *Flange* has to be at the Pose  $F$ , which however is not reachable. In this case, two solutions are possible: The first solution is to make sure that the end effector reaches the position of  $G$ , ignoring the orientation. In this case, a **PositionPreservingProjector** is responsible for projecting the pose  $G$  to  $G_{pos}$  (as a goal for the end effector) or  $F_{pos}$  (as a goal for the flange) as shown in figure 5.7(b). In other cases, the orientation of the goal may be more important. Then, the goal  $G$  is projected to a position  $G_{ori}$  (or  $F_{ori}$ ) that leads to a correct orientation of the end effector, while the given position is not reached, as shown in figure 5.7(c). In this case, an **OrientationPreservingProjector** is used.

### Frame Projectors for the youBot Arm

In the case of the youBot arm, the kinematic constraints are similar: While the youBot arm can reach every point in 3D space, in most positions it cannot rotate around the axis of the first Joint. Due to its kinematic structure, the rotation axis of the fifth Joint and thus the gripper is

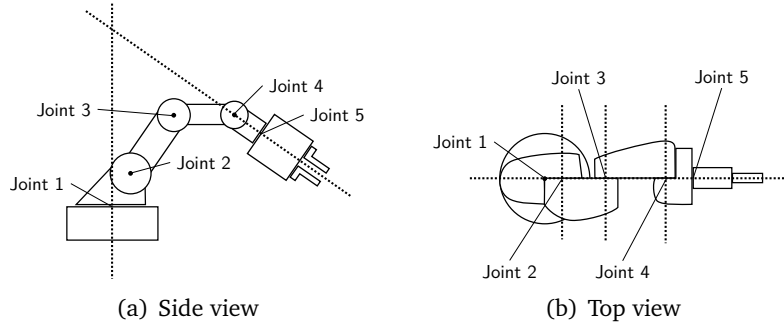


Figure 5.8: Kinematic structure of the youBot arm, forcing the first and fifth axis to intersect or to be parallel

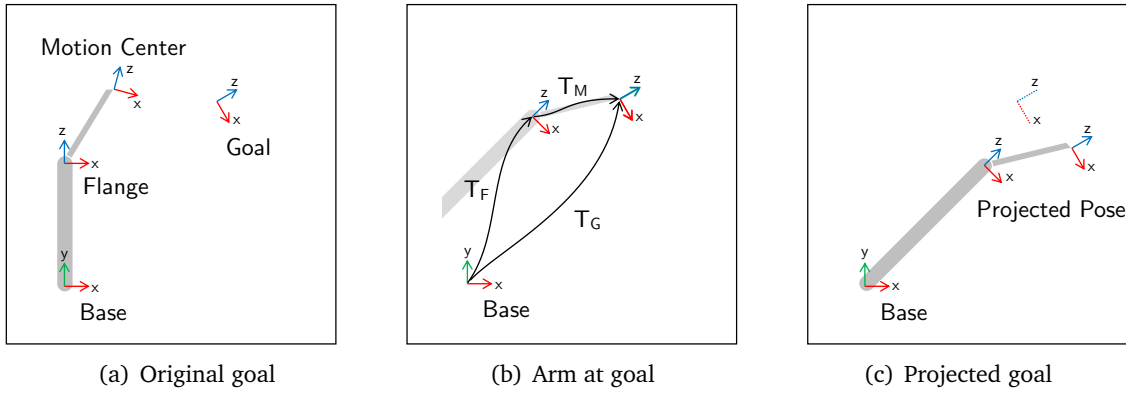


Figure 5.9: OrientationPreservingProjector for the youBot arm

always parallel to the axis of the first Joint or intersects with it (cf. figure 5.8). In this situation, both a PositionPreservingProjector and a OrientationPreservingProjector are available.

The OrientationPreservingProjector is easy to implement, but less useful when only using an arm without platform. It corrects the given Pose so that the new goal has the same Orientation, but the position differs (cf. figure 5.9). The FrameProjector works on a Transformation  $T_F$  that gives the desired Pose of the arm's flange relative to its base, which has been calculated from the goal Pose  $T_G$  and the Transformation  $T_M$  from the flange to the motion center (cf. figure 5.9(b)). If the Transformation  $T_F$  is already valid (i.e. the Z axis of the flange points upwards or downwards, or its projection to the X-Y plane is parallel to the connection between base and flange), no correction has to be applied. Otherwise, the position of  $T_F$  has to be changed so that it becomes valid. Therefore, the X and Y component of  $T_F$  are updated so that their norm in the X-Y plane remains constant, but the direction is given by the X-Y projection of the Z axis of  $T_F$  or its inverse, whichever is closer (cf. figure 5.9(c)). In addition, upper and lower bounds for the norm of  $T_F$  in the X-Y plane can be given with the OrientationPreservingProjector, allowing to approach the orientation of positions that are far outside the reachable space.

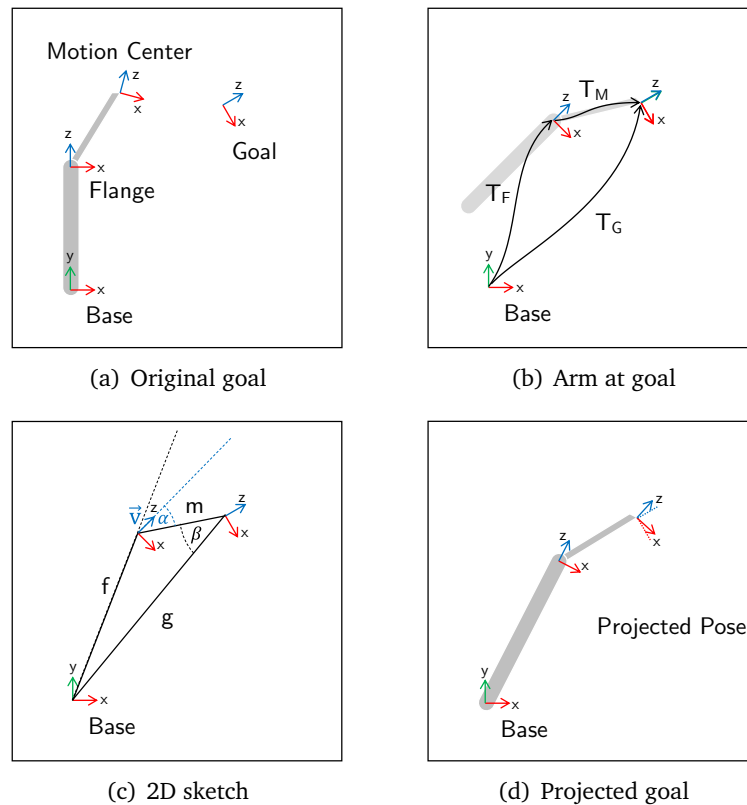


Figure 5.10: PositionPreservingProjector for the youBot arm

In contrast, the PositionPreservingProjector is a bit more complex, and becomes useful when the motion center has to reach a given goal, while the exact orientation is less important. It corrects the given Pose in a way that the position of the configured motion center remains constant, but the orientation is changed (cf. figure 5.10). It takes into account the Transformation  $T_F$  of the flange relative to the base and the Transformation  $T_M$  from flange to motion center (cf. figure 5.10(b)). If the Transformation  $T_F$  is already valid, i.e. its Z axis points up or down, or along the arm, the Transformation is accepted. Otherwise, four possible rotations are calculated that could be applied to the goal so that the Transformation becomes valid: Two rotations to turn the Z axis of  $T_F$  upwards or downwards, and two to turn it so that it points at or away from the base Frame. To rotate the Z axis up or down, the axis is found through the cross product of the flange Z axis with the global Z axis, while the angle can be calculated through a dot product of the two axes. The other two solutions are calculated after projecting the problem into the X-Y plane (cf. figure 5.10(c)). There,  $\vec{f}$  is the 2D vector of the  $T_F$  Transformation,  $\vec{m}$  is the 2D vector of the  $T_M$  Transformation and  $\vec{g}$  is the 2D vector of the goal position, calculated through  $T_F \cdot T_M$ . Additionally, the 2D direction of the flange Z vector  $\vec{v}$  is used from the third column of the  $T_F$  transformation matrix. If  $|\vec{m}|$  is 0 (i.e. the motion center has the same 2D position as the flange), the required rotations around the global Z axis can be calculated as the angle

between  $\vec{f}$  and  $\vec{v}$ , or  $\vec{f}$  and  $-\vec{v}$  respectively. Otherwise, the triangle between  $\vec{f}$ ,  $\vec{m}$  and  $\vec{g}$  has to be inspected (cf. figure 5.10). The goal is to find a rotation of the given Pose around the Z axis, so that  $\vec{v}$  becomes parallel to  $\vec{f}$ . Therefore, the lengths of  $\vec{m}$  and  $\vec{g}$  are assumed to remain equal, while  $\vec{f}$  will be changed. First, the angle  $\alpha$  between  $\vec{m}$  and  $\vec{v}$  is calculated:

$$\alpha = \arccos\left(\frac{\vec{v} \cdot \vec{m}}{|\vec{v}| \cdot |\vec{m}|}\right)$$

This angle is then taken as the expected angle between  $\vec{m}$  and  $\vec{f}$ . Using the law of sines and the sum of angles of a triangle, angle  $\beta$  can then be computed:

$$\frac{\sin(\pi - \alpha)}{\sin(\alpha - \beta)} = \frac{|\vec{g}|}{|\vec{m}|}$$

Comparing the resulting angle  $\beta$  with the one between  $\vec{m}$  and  $\vec{g}$ , the required rotation can be calculated. A similar computation can also be performed to find the angle so that  $\vec{v}$  points in the opposite direction of  $\vec{f}$ . Having the four possible rotations leading to a reachable orientation, the one with the smallest angle is used to correct the goal (cf. figure 5.10(d)).

### Frame Projector for the youBot Platform

For the youBot platform, a PositionPreservingProjector is available to project the given Pose into the X-Y plane. Additionally, it has to change the orientation so it reflects the Z rotation, but still keeps the position of the motion center in the given goal point. In contrast, an OrientationPreservingProjector does not make sense here, because the platform has no position-dependent orientation constraints.

The commanded Cartesian position for the platform can be modified by a PositionPreservingProjector to make it reachable while keeping the defined motion center at the desired position (cf. figure 5.11). The projector works on the desired Transformation of the platform relative to the odometry origin, called  $T_F$ , and the Transformation from the platform position to the motion center  $T_M$ , and performs its work in two steps. First, the Orientation of  $T_F$  is corrected to  $T'_F$ , so that it becomes a feasible orientation for the platform (i.e. it is limited to a rotation around the Z axis) while the motion center is still at the same position. Therefore, the Transformations  $T_G$  (available as the product of  $T_M$  and  $T_F$ ) and  $T_M$  are reduced to a translation in X, Y, and Z and a rotation around Z (setting the X and Y rotations to 0), yielding  $T'_G$  and  $T'_M$ . Then,  $T'_F$  can be computed as the product of  $T'_G$  and the inverse of  $T'_M$ . As a second step, the Transformation is projected into the X-Y-plane by setting the Z coordinate to 0, keeping the orientation constant.

Note that the computations performed by a PositionPreservingProjector or OrientationPreservingProjector depend on the Device used and differ between youBot platform and arm. Additionally, the computations are dependent on the currently selected motion center and cannot be performed based on the flange position alone.

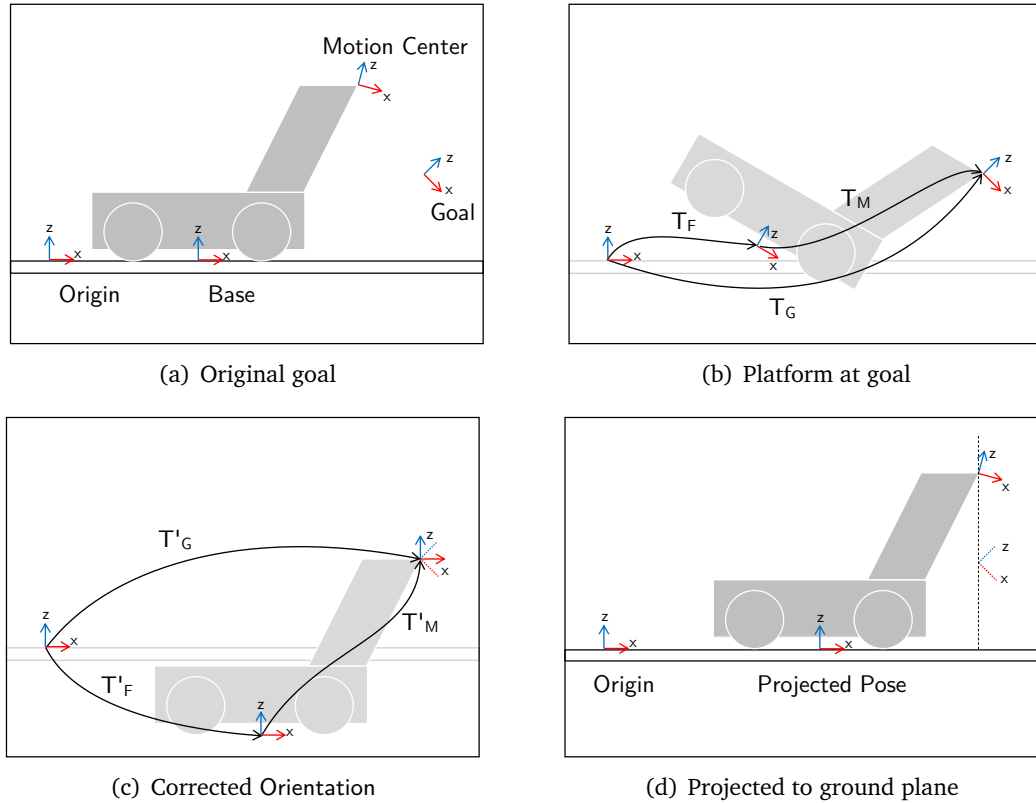


Figure 5.11: PositionPreservingProjector for the youBot platform

### Frame Projector for the Mobile Manipulator

In the case of a youBot platform with arm, a system with 8 degrees of freedom (3 for the platform and 5 for the arm) exists that can reach every position with each orientation (within a given working envelope). Using FrameProjectors, independent motions for platform and arm can be used to approach a given goal. As the platform has more constraints on the orientation than the arm, the arm is used to reach the orientation, while the platform is responsible for the X-Y position. To achieve this, an OrientationPreservingProjector is used on the arm, while the platform works with a PositionPreservingProjector. Both use the gripper as a motion center, and both are commanded to approach the same goal. In this case, the arm will reach a position with the correct Z coordinate and the right orientation, while the platform will contribute the right X and Y coordinate. However, the exact orientation of the youBot after execution depends on the start position of the youBot and the individual speeds of platform and arm, because the motion ends once they have reached their desired goal through interaction. To gain more control, the motions of platform and arm can be executed sequentially, leading to a final position where the arm pose is similar to its initial position.

## 5.3 Comparison with Previous Work on the Robotics API

The general notion of Actions as a description of the *what* aspect of a task independent from the Actuator that will execute them has been taken without modification from Angerer [2] and Schierl et al. [89]. Extending the previous work, this chapter gives a more detailed overview and classification of different motions and actions that can be executed by (mobile but also stationary) robots, following design goal 5.1. However, some concepts were renamed for greater generality or clarity: The distinction between Goal and Path Actions [89] was generalized to Goal and Process Actions, also covering continuous processes that do not belong to a geometric path, e.g. to describe color changes of lamps or speed profiles of an electric screw driver or conveyor belt. *JointMotion* and *CartesianMotion* as introduced by Angerer [2] have been renamed to *JointTrajectory* and *CartesianTrajectory* for consistency reasons, to differentiate them from other motions in Cartesian or Joint space.

Looking at the available *DeviceParameters*, Cartesian and Joint limit parameters along with the Motion center parameter have already been introduced and used by Angerer [2]. However, Angerer [2] suggested that configuring the used controller of the robot should not be implemented as a *DeviceParameter*, because depending on the used robot, switching controllers can take an indeterministic amount of time (at least for KUKA lightweight robots). Still, a set of *DeviceParameters* has been introduced in this work that allows to parameterize controllers (e.g. gains for position control, or stiffness and damping for impedance control) for a motion. These parameters however do not switch between different controller modes, but only parameterize the currently chosen controller, while discrete switching (that may take time on certain robots) is still modeled as a separate Action. This helps with the implementation of Devices that perform control in RPI nets (e.g. the quadcopter that only provides one controller mode and thus cannot switch, however requires the chosen parameters to set up its data-flow graph, cf. section 9.2.2), while still keeping indeterministic parts away from *DeviceParameters*.

Finally, *FrameProjectors* and their corresponding *FrameProjectorParameter* have been introduced as a new concept for handling devices with limited degrees of freedom (which is not required for industrial robot arms with at least 6 joints, but plays an important role with mobile robots). Using them, it becomes possible to specify arbitrary tasks and motions in Cartesian space (cf. design goal 5.1), and to independently configure how the motion is to be executed under the kinematic restrictions present for a given robot (cf. design goals 5.2 and 5.3).

## 5.4 Related Work

Looking at other robot frameworks, different types of motion definition can be found. In *ROS*, motion specification in Joint space is possible through the messages available in the package *trajectory\_msgs* [30]. There, a *JointTrajectory* allows to specify goal positions, velocities and accelerations in Joint space, along with times when the corresponding point should be reached.

Then, an actuator controller is responsible for performing the motion, using spline segments. For mobile platforms, the actuator driver accepts a commanded velocity, which has to be provided by further components such as planners. In *OROCOS* [14], standard motions are defined by instantiating trajectory generator components that provide set-points for actuator driver components. These trajectory generators can work in Cartesian or Joint space and are responsible for providing trajectories that can be followed by the chosen actuator. Similarly, goals can be handled by controller components that perform closed-loop control in conjunction with actuator driver components. However, these ways of motion definition are rather execution-centric, and are not usable for device-independent motion specifications. They can however be seen as a target platform for the execution of Actions as introduced in this chapter.

For higher level motion specification, *ROS* provides planners for robot arm or mobile platform motions. These accept goals in Cartesian or Joint space, and create trajectories with feasible velocities, accelerations and times for a given actuator. For robot arms, planners such as *arm\_navigation* [50] or since 2012 *Movel* [17, 97] are used. They create a *JointTrajectory* based on a given robot and goal, while avoiding collisions and optionally following further constraints (e.g. forcing the end effector orientation to remain constant). For mobile platforms, motion planning, obstacle avoidance and trajectory tracking are handled in a planner called *move\_base* [69] that accepts a goal pose in Cartesian space. However, defining and following a predefined motion in Cartesian space is not directly possible using these standard means, but requires specialized components. While allowing goals that can be given in an actuator-independent way and applied to different robot arms, these planners rather correspond to the device capabilities introduced in chapter 7. In contrast, the Actions introduced in this thesis can be seen as an intermediate layer, allowing to define device-independent operation specifications with clearly given semantics that can be executed on compatible Actuators. As this, they fulfill design goal 5.1, which is not handled to a similar extent in the compared robot frameworks.

As an alternative to giving a single goal or a complete trajectory for a robot, constraint-based methods are sometimes used, especially for redundant robots. These include *SoT* (Stack-of-Tasks, [67]) in the *ROS* ecosystem and *iTaSC* [23, 101] for *OROCOS*. The idea is to give the desired trajectory through constraints and cost functions, while seeing motion execution as an optimization problem. *SoT* is meant for redundant robots and allows to give multiple levels of constraints and tasks. The constraints give boundaries that may not be violated, while the optimization criterion is responsible for leading the task towards progress. In the task stack, lower-priority tasks are executed in the nullspace of high-priority tasks, i.e. the degrees of freedom that are redundant or do not contribute to the high-priority task can be moved in a way so that the low-priority tasks are fulfilled or approached.

*iTaSC* [101] also allows to define tasks through constraints, but adds estimation to handle uncertain sensor data. Task specifications consist of virtual kinematic chains (that describe the environment and the relationship between robots and objects and support forward and



inverse kinematics and dynamics), of constraint-outputs and constraint-controllers to define and control the constraints, of set-point generators that give desired values for controllers, and of an estimator for uncertain state. Here, set-point generators are closest to the Action concept, while estimators correspond to the estimation process described in chapter 8.

These constraint-based approaches can be helpful for redundant robots or for tasks that do not constrain all degrees of freedom. For example, the combination of youBot arm and platform can be used to execute tasks in Cartesian space, while further constraints give hints for the desired state of the redundant degrees of freedom (e.g. the platform rotation). Alternatively, a task that only describes the position of the end effector but does not give any constraints for the orientation can be used instead of a PositionPreservingProjector to make the end effector reach a given position in space.

This way, they are an interesting extension to the approach proposed in this thesis. During the work on this thesis, a first proof of concept version has been implemented for multiple robots, allowing to constrain properties of geometric relationships or joint positions while performing motions in other degrees of freedom, however only in pure Java without real-time guarantees. This kind of behavior could be offered as further Actions for more complex motion specifications (while using the existing Actions as set-point generators). However, due to the higher computation load of the constraint-optimization problems, their use can also be problematic in environments with limited processing power, where explicit motion specification (optionally with FrameProjectors) can solve many tasks with less computational overhead.

For treating the youBot as a redundant mobile manipulator, instead of using an OrientationPreservingProjector for the youBot arm and a PositionPreservingProjector for the platform, both can be combined and handled as a single kinematic structure. Then, a specialized closed-form inverse kinematics function can be used, as suggested by Sharma et al. [94]. For a given position and orientation in Cartesian space, this inverse kinematics function provides a position and orientation of the youBot base, along with Joint angles for the arm. To define the redundant degrees of freedom, the function introduces three parameters  $\rho_1$ ,  $\rho_2$  and  $\rho_3$ . The rotation of the platform relative to the arm is described by  $\rho_1$  (the platform can compensate for rotations of the first arm Joint), while the distance of the platform to the goal is given by  $\rho_2$  (where Joints two to four can be used to position the end effector at the desired position), and  $\rho_3$  decides whether to use an elbow-up or elbow-down configuration.

Using the FrameProjector approach,  $\rho_2$  corresponds with the parameter for the desired distance used with the OrientationPreservingProjector, while the parameters  $\rho_1$  and  $\rho_3$  cannot directly be influenced when combining motions of platform and arm. The binary parameter  $\rho_3$  is already chosen by the initial configuration of the arm and cannot be changed during a Cartesian motion (because this would require a non-continuous motion of the arm), while  $\rho_1$  depends on the initial configuration and the motion synchronization of platform and arm and emerges from the interplay of the two FrameProjectors. However, using independent arm and platform

motions with their `FrameProjectors` allows to solve the inverse kinematics problem with given free parameters: First the youBot arm is moved into a position with the desired  $\rho_1$ ,  $\rho_2$  and  $\rho_3$  parameters (e.g. by defining a Cartesian goal with distance  $\rho_2$  in the direction of  $\rho_1$  and using a `PositionPreservingProjector` to find a valid Orientation). Then, a `PositionPreservingProjector` is used with the platform to move to the desired goal, and finally an `OrientationPreservingProjector` for the arm allows to reach the desired goal Pose while keeping the redundancy parameters  $\rho_1, \rho_2, \rho_3$  constant. As an alternative, the inverse kinematics functions of the arm and platform using the corresponding `FrameProjectors` can be used to perform this computation offline without moving the robot, using `FrameTopologys` to describe the intermediate robot positions for the following kinematics function calls.

In comparison, using a combined kinematics function simplifies accessing the redundant degrees of freedom at the cost of a specialized solution, while the `FrameProjector` approach is more generic by combining compositional kinematics functions for the individual devices. For the case study however, platform and arm were controlled individually using `PositionPreservingProjectors` for platform and arm and ignoring the exact gripper orientation (as it does not matter for the round baton grasped), so neither of the approaches was used.

## REAL-TIME COMMANDS AND EXECUTION

After defining geometry and motions, further steps are required to execute them on real hardware. In this context, Requirement 3 requests exact motion execution and precise triggering of Actions. From this requirement, more detailed design goals have been derived. To interface hardware and provide high motion precision, task execution requires a part that is executed in real-time, i.e. with the guarantee that certain time bounds will not be violated.

**Design Goal 6.1.** *Define tasks to be executed with real-time guarantees.*

To be able to execute multiple tasks without unintended delays, it also has to be possible to combine multiple tasks so that also their transition happens with real-time guarantees.

**Design Goal 6.2.** *Create combinations of tasks that will be executed with real-time guarantees.*

On mobile manipulators, this is important for their different parts, e.g. platform and arm, where synchronization can be required to achieve the desired overall behavior. When Actions are to be triggered by certain events, a way to specify trigger conditions is required, based on data available at run time.

**Design Goal 6.3.** *Handle time-variable data within real-time contexts.*

This data should then be usable with real-time guarantees to trigger new Actions, but also to affect running Actions, either in a discrete (switching) or in a continuous (control) manner.

**Design Goal 6.4.** *Allow the use of time-variable data to affect task execution.*

This is required for mobile robots in less structured environments, when they have to react to sensor data or environment changes. Additionally, sensor data about the other parts of a

mobile manipulator may be required for the task of a certain device, e.g. the current position of the platform when trying to move the arm to a position fixed in the environment. Finally, the running application should also gain access to run-time data, for visualization or application-specific planning purposes, allowing to perform planning based on the perceived surroundings of a mobile robot.

**Design Goal 6.5.** *Provide access to time-variable data or events in an application context.*

Following these design goals, elementary tasks for Actuators are not modeled as methods on the Actuator objects, but rather through Command objects following the command pattern [31]. A Command brings together an Action with an Actuator and further parameters, cf. section 2.1.2. **Commands** describe execution units with real-time guarantees, which can be defined and started from a non-real-time environment, but also linked to each other to execute composed real-time tasks.

For execution, Commands are transferred to an **execution environment**, where they are evaluated with real-time guarantees, commanding the Actuator according to the specified Action. In the proposed software architecture, the Robot Control Core serves as an execution environment that can execute Commands that have been converted to Data-flow graphs.

In section 6.1, details are given how motion specifications, Actuators and further properties are linked to define complete task specifications, while section 6.2 goes into details about how these task specifications are converted to data-flow graphs for execution on the Robot Control Core. Finally, the chosen solution is compared to previous work on the Robotics API (section 6.3) and further approaches used in mobile robotics (section 6.4).

## 6.1 Commands Linking Actions to Actuators

Within a Command, the Action is responsible for giving set-points for the desired behavior (the *what* dimension), while the Actuator is responsible for interpreting and following the set-points (the *who* dimension) and respecting the DeviceParameters giving concrete details about the execution (the *how* dimension). Additionally, rules can be given that define *when* to execute the Command. Within a Command, time-variable data is used in the form of RealtimeValues (cf. section 6.1.1), defining computations that are to be executed with real-time guarantees.

Section 6.1.2 describes inner structure of Commands consisting of Actions, Actuators, DeviceParameters, and further elements. Section 6.1.3 explains the available operations of Commands and their life-cycle.

### 6.1.1 Realtime Values – Working with Time-Variable Data

To facilitate the specification of time-variable data as description of computations, a set of new data types has been introduced, namely the class hierarchy of **RealtimeValue**. The specialized

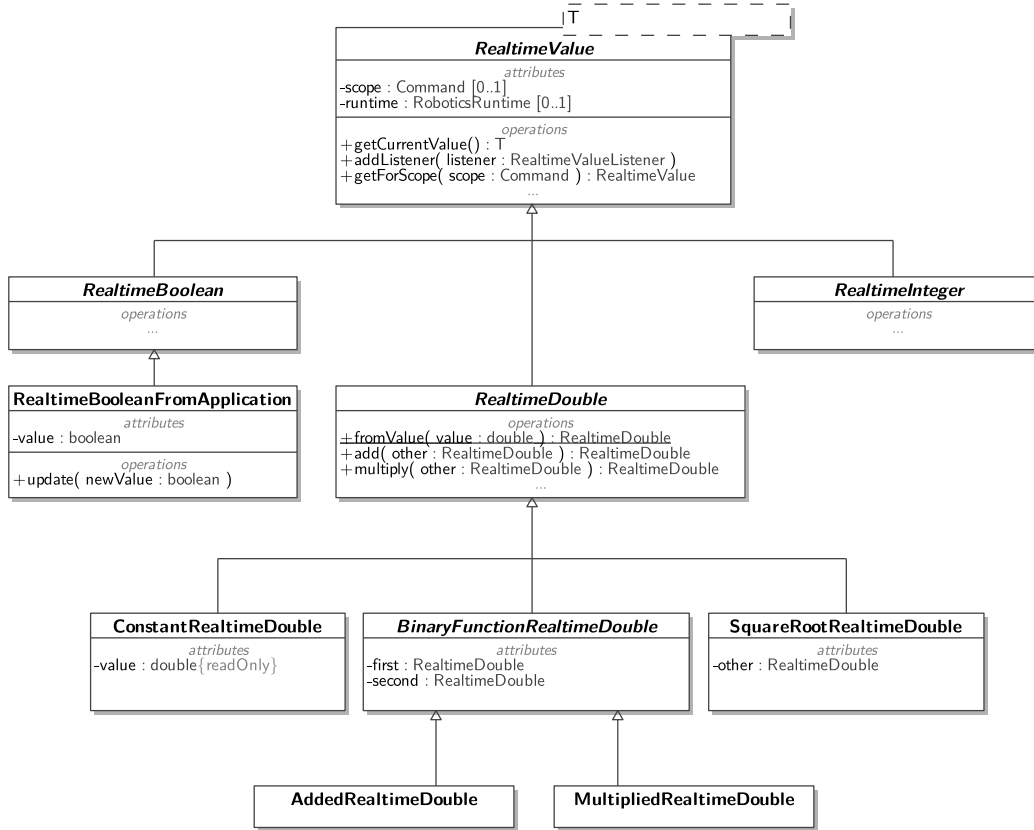


Figure 6.1: RealtimeValues for specifying computation laws

types of `RealtimeValues` represent a value of the corresponding data type available in real-time in an execution environment, and allow to execute operations on them. However, the operations are not immediately applied to the values, but stored in a tree data structure, so that they can later be translated to the target language of the execution environment where they are evaluated to compute the value.

Figure 6.1 introduces some of the main types of `RealtimeValues`, along with some of the frequently used operations. Specialized `RealtimeValues` are provided for primitive data types (e.g. **`RealtimeBoolean`**, **`RealtimeDouble`**), as well as for composed data types (e.g. **`RealtimeVector`**, **`RealtimeTransformation`** and **`RealtimeTwist`**), semantically enriched data types (e.g. **`RealtimePoint`**, **`RealtimePose`**, **`RealtimeVelocity`**) and array types (e.g. **`RealtimeDoubleArray`**). Each has one subclass to provide a constant value as a `RealtimeValue`, e.g. **`ConstantRealtimeDouble`** and **`ConstantRealtimeInteger`**. Furthermore, for each of the available operations (unless it is simply a shortcut for another operation), a subclass of the typed `RealtimeValue` exists that represents the operation and holds references to the operands. Binary functions such as multiplication of floating-point numbers are thus modeled as subclasses of **`BinaryFunctionRealtimeDouble`**, such as **`MultipliedRealtimeDouble`** and **`AddedRealtimeDouble`**. Additionally, time-variable val-

ues from an application can be provided without real-time guarantees through classes such as **RealtimeDoubleFromApplication** and **RealtimeIntegerFromApplication**.

At run time, each *RealtimeValue* represents a value of the given data type, or the value *null* if no value is currently available. The calculation operations generally work on *null* values in a strict fashion, returning *null* if any of the required input parameters is *null*. An exception is made for purely *Boolean* calculations, where three-valued logic is used and *null* is returned when the result is unknown for the given input parameters.

Some *RealtimeValues* belong to a *Command* and are only available within this context. In this case, they know the corresponding *Command* as their scope, and can thus only be used within the *Command*. Examples for scoped *RealtimeValues* are the progress of an *Action* or the elapsed execution time of a *Command*, which can only be used for computations within the *Command* (e.g. to find the right set-point for the given time).

To improve the performance of evaluation, a set of optimizations is used on *RealtimeValue* calculations to reduce the computation overhead on the execution environment:

- The first optimization is to perform constant folding (cf. [74]). When applying a computation to a set of constant values, the result is still constant and can thus be provided as constant, without evaluating the computation in each time step.
- A similar optimization is used when applying the neutral element – adding 0, multiplying with 1 or applying a unit matrix. These operations have no effect and can thus be ignored.
- The application of inverse operations can be omitted, e.g. when negating a floating point value or inverting a transformation matrix twice.

One use case for *RealtimeValues* is the youBot soft gripper (cf. section 4.2.1): Through the closed kinematic chain, the linear motion of the gripper jaws is transformed into a circular motion of the fingers. This correlation can be described through *RealtimeValues*, working on a *RealtimeValue* for the finger distance and applying the formula described in section 4.2.1 and figures 4.14(b) and 4.15 to provide *RealtimePoses* of the finger base *Frames*. These *Poses* are then used in *Relations* linking the fingers to the gripper, thus providing the correct geometry of the gripper. Furthermore, *RealtimeValues* are used for motion definitions, as explained in section 6.2.1, and can also be used in applications as described in chapter 10.

### 6.1.2 Commands – Executable Specifications for Actuators

*RealtimeValues* are the way to reference data available at run-time in robot tasks modeled as *Commands*. Figure 6.2 gives an overview of the concepts related to *Commands*. As main parts, a *Command* consists of an *Action* along with the *Actuator* and a set of *DeviceParameters*. Additionally, it contains *CommandResults* and *Assignments*.

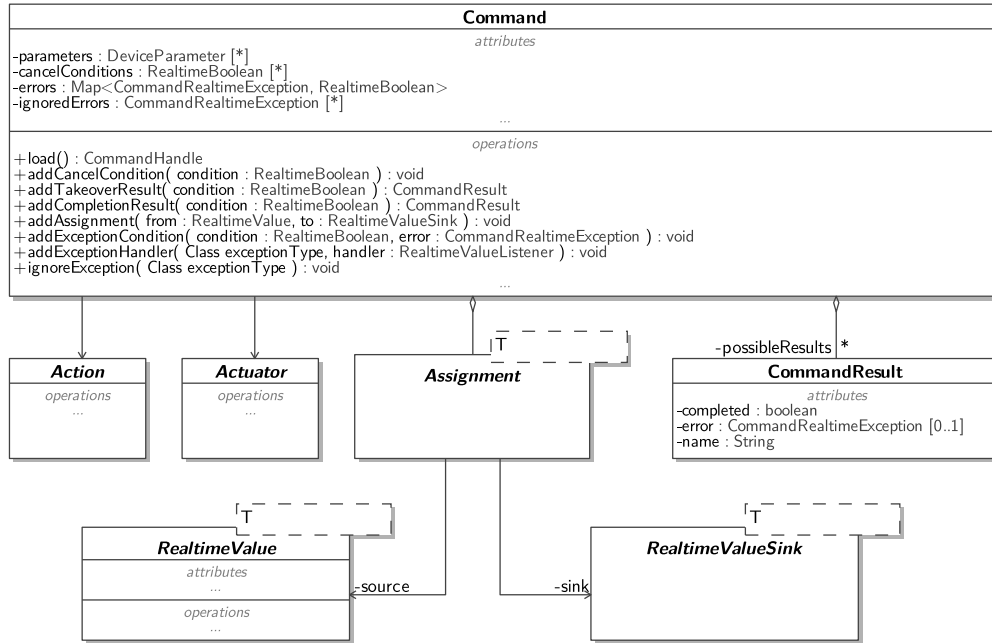


Figure 6.2: Class diagram of concepts related to Commands

A **CommandResult** describes one possible outcome of the Command execution. Some CommandResults signal the completion of the Command, while others represent errors that occurred in the Command. Furthermore, interesting situations that do not require the Command to end but may be used to switch to another Command can be modeled as CommandResults. The occurrence of CommandResults is defined through the methods **addCompletionResult()** and **addTakeoverResult()** accepting a **RealtimeBoolean** (cf. section 6.1.1) describing the situation when the CommandResult is active. Similarly, **addCancelCondition()** accepts **RealtimeBooleans** (expressing certain situations in the Command) to define conditions when the Command should be canceled.

By default, the Command is completed once both Action and Actuator signal completion. An Action signals completion once it has provided its final set-point (as described in section 5.1), while the Actuator signals completion whenever the given set-point has been reached. Through **addCompletionResult()**, further situations can be defined when the Command has completed.

Apart from the main task of executing an Action with an Actuator, a Command can contain **Assignments** that assign a **RealtimeValue** available during Command execution to a user-defined data sink (**RealtimeValueSink**, cf. figure 6.3). Assignments are added to a Command through the **addAssignment()** method. One example of **RealtimeValueSinks** is the **RealtimeValueReporting** which allows to add a **RealtimeValueListener** to a **RealtimeValue** to observe sensor data or internal states. This listener is part of the application and is informed about the current value without real-time guarantees, but still allows the application to react to certain conditions. Further **RealtimeValueSinks** are **ParameterAssignments** which allow to change Device parameters

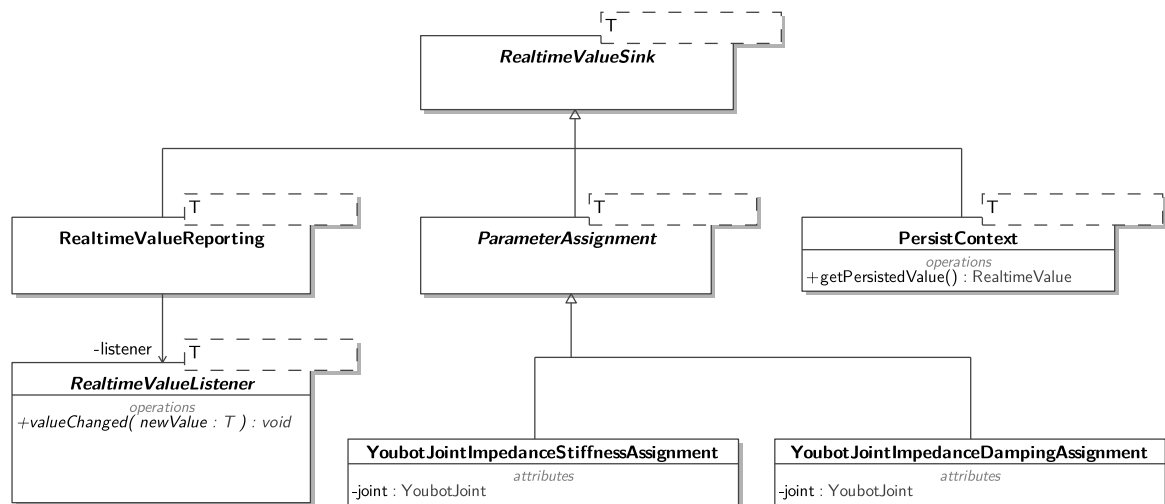


Figure 6.3: Typical types of RealtimeValueSinks

such as the youBot arm stiffness during the execution of a motion, as well as **PersistContexts** that persist RealtimeValues, thereby making the corresponding RealtimeValue available to other Commands during the lifetime of the Command that contains the Assignment. PersistContexts are especially helpful to make RealtimeValues that have a limited scope available in other Commands, or if a complex and time consuming RealtimeValue computation is required in multiple Commands.

Furthermore, errors can be defined for a Command through **addExceptionCondition()**. These errors (that have to be instances of subclasses of **CommandRealtimeException**) occur when the given condition (given as a RealtimeBoolean) becomes *true*. Defined errors – selected either by their class or instance – can be handled by **addExceptionHandler()** through a RealtimeValueListener, or by ignoring them using **ignoreException()** (if they are no problem in the given situation). Unhandled errors lead to error CommandResults that immediately abort the Command and cause the corresponding CommandRealtimeException to be thrown in the application.

### 6.1.3 Command Operations and Scheduling Rules – Controlling Commands

A Command can be loaded through the **load()** method, which sends it to a real-time execution environment. The execution environment is modeled as a **RoboticsRuntime** that serves as a proxy (cf. [31]) to access sensor data and execute Commands. Loading a command returns a **CommandHandle** that allows to further influence execution (cf. Figure 6.4). The CommandHandle supports various operations:

**start()** requests the execution environment to begin the execution of the loaded Command. The operation waits for the Command to start, and returns an error if starting fails.



## 6.1. COMMANDS LINKING ACTIONS TO ACTUATORS

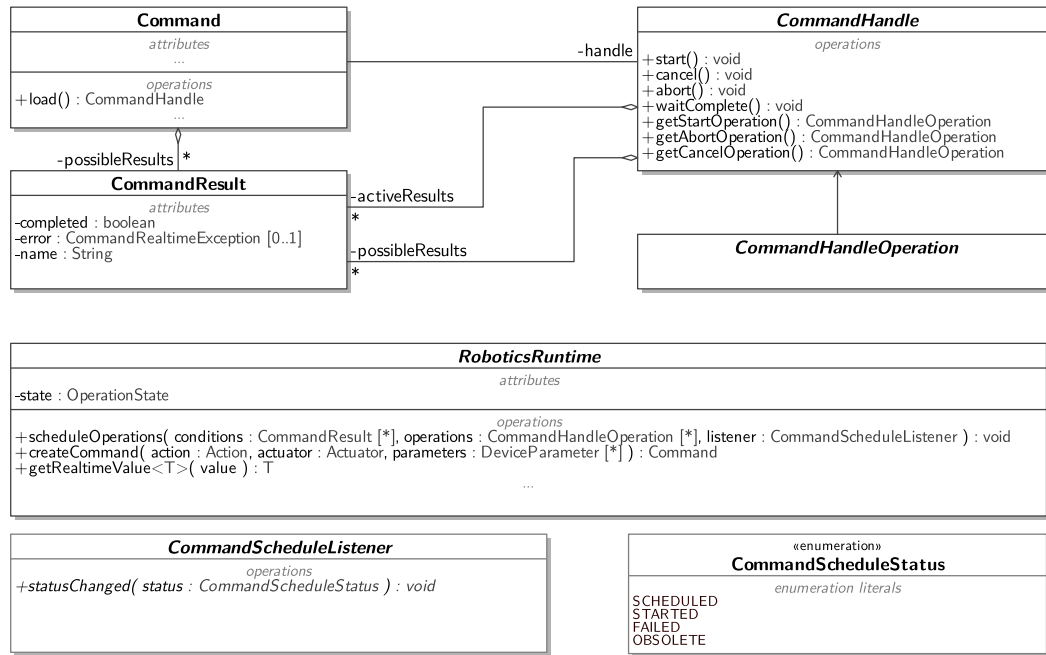


Figure 6.4: Class diagram of concepts related to the execution of Commands

**cancel()** requests the Command to gracefully stop the current operation. The Command can then decide internally how to handle the request, ideally leading to a situation where a completed CommandResult is reached earlier than at normal execution. Both Action and Actuator can react to **Cancel** request to quickly bring the Actuator into a safe state.

**abort()** forcefully terminates the execution. This does not bring the controlled Actuators into a stable state, and should thus only be used if the situation is safe or a successor can handle the current situation (see below).

**waitComplete()** blocks the workflow until any completed CommandResult occurs and the Command is thus terminated.

To get additional information about the execution, CommandHandles allow to list the CommandResults that are currently active, as well as the CommandResults that can still occur. However, all these operations only allow applications to influence Commands without real-time guarantees.

It is also possible to define reactions that will be executed in real-time: The execution environment allows to schedule Command operations that will be executed under given conditions. Such a schedule consists of a set of CommandResults as a trigger condition, as well as a set of **CommandHandleOperations**. The CommandResults may belong to different Commands, and are interpreted as a conjunction, triggered when all of the given CommandResults are active at the same time. CommandHandleOperations can be to start, cancel or abort any previously loaded CommandHandle. Typically, such schedules are used to take over running Commands once they

reach a given condition, aborting the running Command and starting a new one. The schedule is defined without real-time (i.e. issuing the definition can take an arbitrary amount of time), but the specified operations are executed in real-time once the given condition occurs, stopping one Command and starting the next without skipping an execution cycle of the real-time system. To be able to monitor such schedules, the **scheduleOperations()** method accepts a listener that is informed whether the schedule did already occur or not. Additionally, it notifies when the operations could not be executed although the precondition was met because any of the Commands was not in an acceptable state, or if the schedule can no longer occur because the preconditions have become impossible.

Using Commands and scheduled command operations, it is possible to define complex behavior of robots, as explained in the following chapters.

## 6.2 Executing Actions – From Commands to Real-Time Control

To be able to execute tasks, an execution environment is required that communicates with the hardware and is able to provide sensor data and to run Commands, forwarding application-defined set-points from Actions to a given Actuator. Therefore, the execution environment requires a flexible language to specify the expected behavior, along with the low-level Device drivers required to execute the specification. In the reference implementation based on the work of Vistein [103], the execution environment is called Robot Control Core and can be controlled through Data-flow graphs and Synchronization rules specified using the **Realtime Primitives Interface** (RPI). There, Data-flow graphs describe continuous behavior by specifying how data from sensors and calculations is propagated to actuators, while Synchronization rules define points for switching to other Data-flow graphs (cf. section 2.1).

Data-flow graphs are modeled as object structures (cf. figure 6.5), and consist of calculation Primitives with input and output ports (**InPort** and **OutPort**) as well as **Parameters**. To form a graph, an InPort can be connected to an OutPort of another Primitive to forward the resulting value. Additionally, it is possible to combine multiple Primitives with their corresponding connections into a Fragment, still with input and output ports to be connected from outside (**FragmentInPort** and **FragmentOutPort**). The resulting Data-flow graphs have to be acyclic (unless *Pre* Primitives are used that delay the data by one execution step and must appear in every graph cycle), so that by topological ordering data can flow through the entire graph in one execution step by evaluating each Primitive once, and all computations are performed on the latest values.

To convert a Command into an executable Data-flow graph, an automatic transformation process is used (as published in [89]). It converts the Action into an executable form (cf. section 6.2.1), and builds a complete Data-flow graph based on the results (cf. section 6.2.2). For this Data-flow graph, Synchronization rules are created to coordinate execution (cf. section 6.2.3).

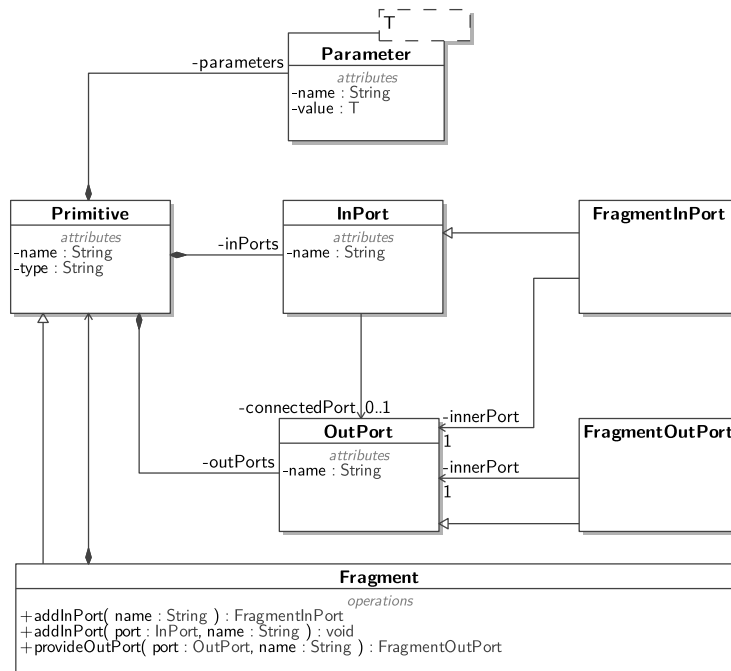


Figure 6.5: Concepts involved in Data-flow graphs (adapted from [103])

### 6.2.1 Making Actions Executable

To execute an Action on a real robot, it is evaluated and provides a set-point including its semantics for each time instant during execution (typically every 2 ms to 12 ms). This can be done in two different ways:

- For the Action, a list can be pre-computed that holds one set-point for each time instant. This way, the complete trajectory is described, and while computing the list, no functional or time limits are in place. In particular, this computation can be performed in the application without restrictions imposed by real-time requirements. However, using this method, there is no way to further influence the trajectory at run time, e.g. to include sensor data.
- The Action can provide *RealtimeValues* (cf. section 6.1.1) that describe how to calculate the set-points. The computations take into account the current time progress, as well as further factors, e.g. sensor values, and are then evaluated on the execution environment whenever a set-point is required, using the current time and sensor data. Within the specified computation, the available arithmetic operations and the allowed execution time are limited. However, during the creation of the specification no real-time constraints are in place, so that all features of the application programming language can be used.

The motion definitions explained in chapter 5 make use of the second method to fully

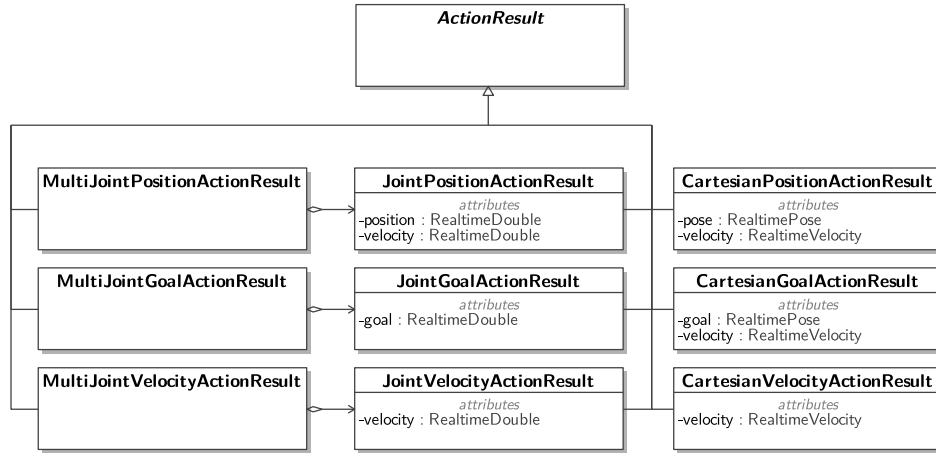


Figure 6.6: Typical ActionResults for motions

support sensor guided motions, however some planning is performed in the application to find parameters for use in the computations. For a given Action, an **ActionResult** is created using a RealtimeDouble for the time progress. The ActionResult provides getters for Realtime-Values representing the computations describing the set-point, and defines the semantics of the set-point (cf. figure 6.6). As an example, the **CartesianPositionActionResult** is used to describe a CartesianTrajectory and provides a RealtimeValue for the position and velocity of the set-point. When working in configuration space, a **JointPositionActionResult** is used to give the desired position and velocity of a single Joint, while a **MultiJointPositionActionResult** can give synchronized set-points for multiple Joints. Working with Goal Actions, **CartesianGoalActionResult**, **JointGoalActionResult** and **MultiJointGoalActionResult** can be used, giving a set-point that is to be interpreted as goal to be reached by an Actuator (but not necessarily in the next time step). For Actions in velocity space, **CartesianVelocityActionResult**, **JointVelocityActionResult** and **MultiJointVelocityActionResult** are used to define that the given set-point is meant as a velocity.

Looking at a simple rectilinear motion (LIN) without orientation change and without jerk limitation, a RealtimeValue for the position is created that takes the current time  $t$  as an input and provides a Pose  $x$  as an output. The computation law consists of three phases combined through a conditional operator:

- For small values of  $t$  ( $0 \leq t \leq t_a$ ), the pose follows a constant acceleration formula in the form of  $x = x_s + 0.5 \cdot a_{max} \cdot t^2$ .
- In the middle ( $t_a < t < t_a + t_c$ ), constant velocity is applied, so that the formula looks like  $x = x_a + v_{max} \cdot (t - t_a)$ .
- Towards the end ( $t_a + t_c \leq t \leq t_a + t_c + t_a$ ), constant deceleration is applied, in the form of  $x = x_c + v_{max} \cdot (t - t_a - t_c) - 0.5 \cdot a_{max} \cdot (t - t_a - t_c)^2$ .

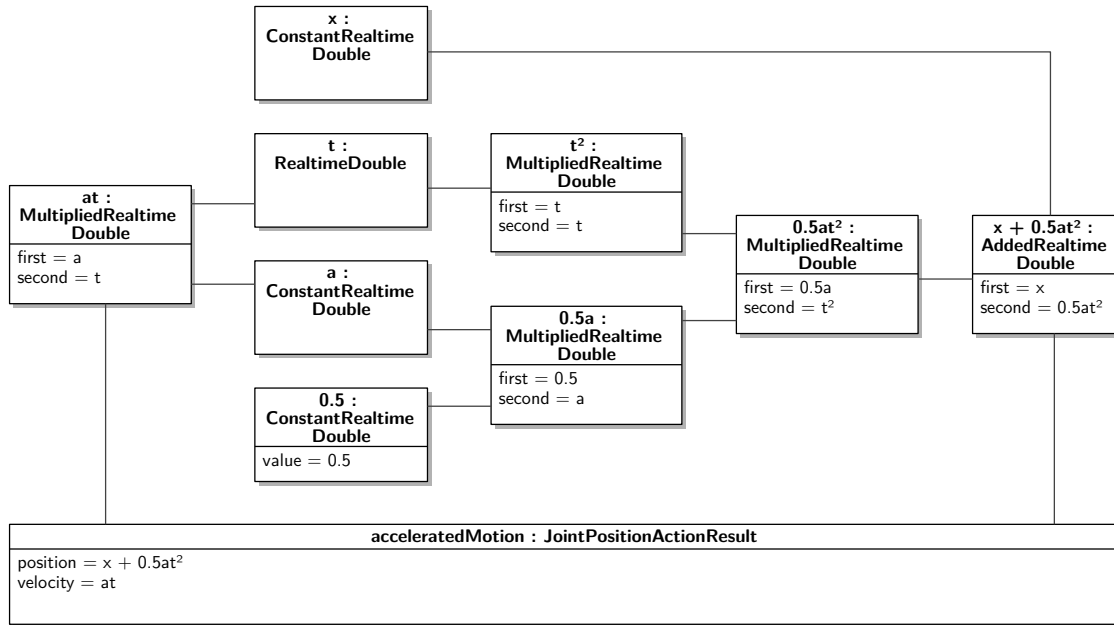


Figure 6.7: ActionResult for an accelerated motion of one Joint

```

1  // Creating an action result for an accelerated motion ...
2  public ActionResult getAccelerationResult(RealtimeDouble t) {
3      // Compute position and velocity ...
4      RealtimeDouble pos = x.add(a.multiply(0.5).multiply(t.square()));
5      RealtimeDouble vel = a.multiply(t);
6      // ... and return the result ...
7      return new JointPositionActionResult(pos, vel);
8  }

```

Figure 6.8: Java code to create a JointPositionActionResult for an accelerated motion

For a given motion, the unknown variables  $t_a$  (acceleration time),  $t_c$  (constant velocity time),  $x_a$  (position after acceleration) and  $x_c$  (position after constant velocity time) have to be calculated in the application. Therefore, the start ( $x_s$ ) and end pose ( $x_e$ ), as well as the allowed maximum velocity ( $v_{max}$ ) and acceleration ( $a_{max}$ ) are used, solving for a minimum time  $t_e = t_a + t_c + t_a$ . For a given `RealtimeDouble t` expressing the time progress, a `RealtimePose` with its metadata (Reference Frame and Orientation) is created and stored in the `CartesianPositionActionResult`. Similarly, the `RealtimeVelocity` is computed and provided (with reference Frame, Orientation and Pivot point).

Figure 6.7 gives an example of the object structure created for the `JointPositionActionResult` of an accelerated motion (i.e. the first part of a PTP motion). Based on the `RealtimeDoubles t` as time progress and  $a$  as acceleration, the velocity is calculated as  $a \cdot t$  through a `MultipliedRealtimeDouble`, while the position is given through  $x + 0.5 \cdot a \cdot t^2$  using `AddedRealtimeDouble`.

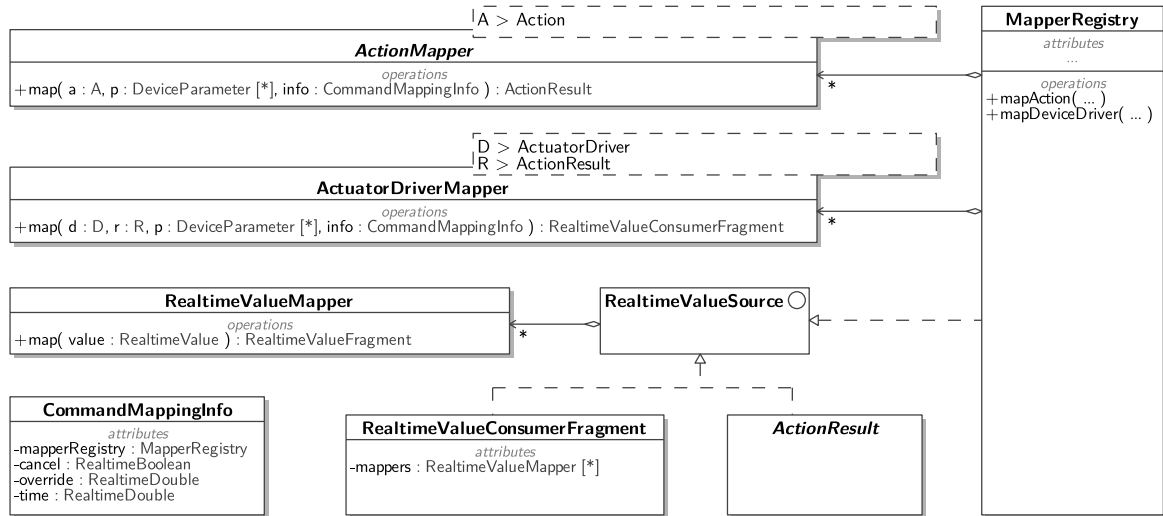


Figure 6.9: Concepts involved in transforming Commands into Data-flow graphs

and MultipliedRealtimeDouble. While the instance diagram looks relatively large, the calculation methods provided by RealtimeDouble allow to create the position term in a straightforward fashion, as shown in figure 6.8. For a complete PTP motion, this result has to be extended by a constant velocity and deceleration phase, separated using *ConditionalRealtimeDouble*.

## 6.2.2 Converting Commands into Data-Flow Graphs

To convert a Command into an executable form, its individual parts have to be converted into data-flow representations and combined into a Data-flow graph. Therefore, different types of **Mappers** are used to transform the different elements present in a Command (over some intermediate steps) into a data-flow representation. This includes converting the Action into a source of set-points as introduced in the previous section, as well as converting the Actuator into a consumer of set-points provided by the Action.

The RoboticsRuntime manages its known Mappers for Actions and Actuators as well as for Sensor data and RealtimeValue computations in the **MapperRegistry** (cf. figure 6.9) To make a Command executable, first the Action together with its configuration in DeviceParameters is transformed into an ActionResult. This happens in an **ActionMapper** which is specific to the used RoboticsRuntime and uses the data stored in the Action as well as the DeviceParameters to build the corresponding ActionResult. As a next step, an **ActuatorDriverMapper** is requested to provide a RealtimeValueConsumerFragment for the given ActionResult and the ActuatorDriver of the corresponding Actuator. A **RealtimeValueConsumerFragment** (cf. figure 6.10) represents a data-flow Fragment, together with the specification of dependencies, i.e. the information that some of the FragmentInPorts expect the values of RealtimeValues or given OutPorts.

To build a complete, self-contained Fragment or Data-flow graph, the RealtimeValues used

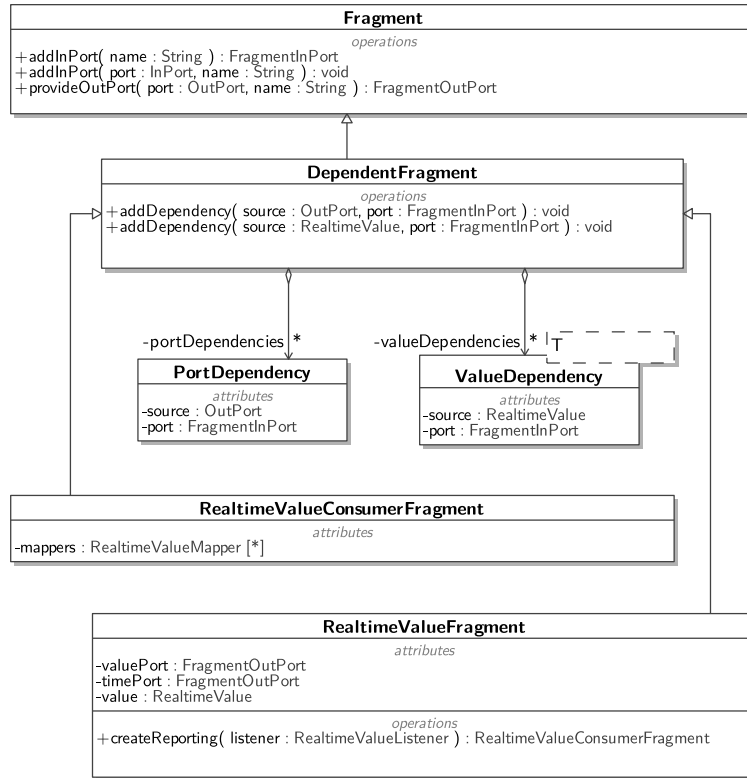


Figure 6.10: Fragments used in the transformation

as dependencies have to be transformed and connected to the specified input ports. Therefore, a **RealtimeValueMapper** is used for each mentioned **RealtimeValue**, which creates a **RealtimeValueFragment** that calculates the current value of the **RealtimeValue**. In contrast to **ActionMappers** and **ActuatorDriverMappers**, **RealtimeValueMappers** are not only provided by the **RoboticsRuntime**, but also by **ActionResults** and **RealtimeValueConsumerFragments**. For example, the information that a given set-point for an **Actuator** is invalid is only available in the **RealtimeValueConsumerFragment** of the corresponding **ActuatorDriver**, so a **RealtimeValueFragment** for a **RealtimeBoolean** representing this condition (to define a **Command** error) can only be provided by this **Fragment**.

**RealtimeValueFragments** can use the available calculation **Primitives**, but are also allowed to define further dependencies to other **RealtimeValues** they require for their computation. For example, a **RealtimeDoubles** representing a **Sensor** value is transformed into a calculation **Primitive** that accesses the sensor and provides its value as an **OutPort**, together with a timestamp representing the age of the sensor data. An **AddedRealtimeDouble** is transformed into a **DoubleAdd** primitive to perform the computation, together with the specification that the two inputs of the **DoubleAdd** primitives are to be connected to the **RealtimeValues** representing the operands.

Some **RealtimeValues** however do not have a universally valid calculation. As an example, talking about the progress of a motion **Activity** shows this problem: When using motion blending

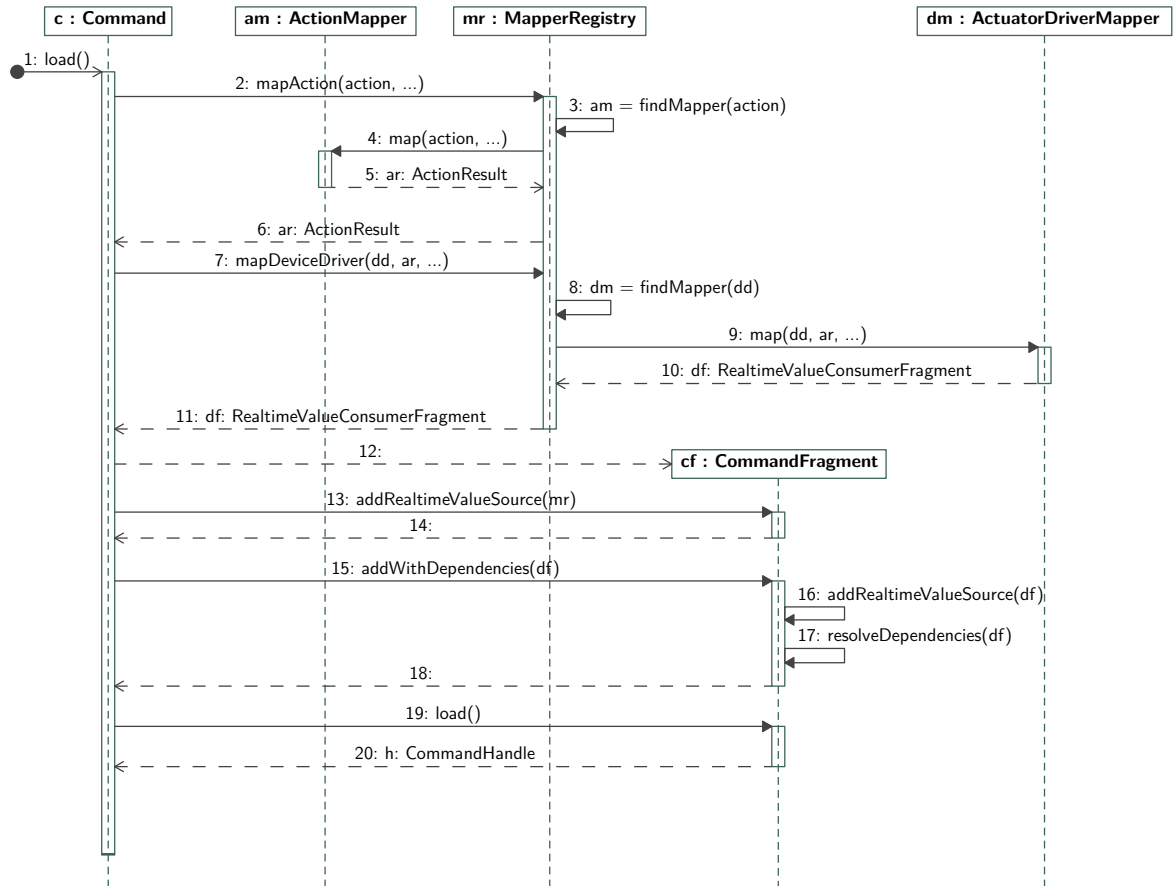


Figure 6.11: Example sequence for transforming a Command into a data-flow graph

(cf. section 7.2.1), two Commands are created for the motion, one with the normal trajectory, and one with a motion that continues the previous motion without stopping. For those two motions, the `RealtimeBoolean` representing a 50% progress has a different meaning. In this case, the method `getForScope()` in `Command` is used to retrieve a `RealtimeValue` specific to the given `Command` before transforming it into a data-flow fragment.

Additionally, `RealtimeValueFragments` for the Cancel request of a `Command` as well as for the elapsed time of `Command` execution are directly provided in the transformation process. For cancel, the `Cancel` primitive is used which is informed if the Data-flow graph is externally canceled, while the execution time is calculated through a `Clock` primitive that counts the seconds since the start of the `Command`.

To build the resulting Data-flow graph for a `Command`, the `RealtimeValueConsumerFragment` for the `ActuatorDriver` together with all the `RealtimeValueFragments` for dependencies is added. As an optimization, duplicate `RealtimeValueFragments` for the same `RealtimeValue` can be omitted, because due to their semantics, the value of one `RealtimeValue` in a given execution cycle has



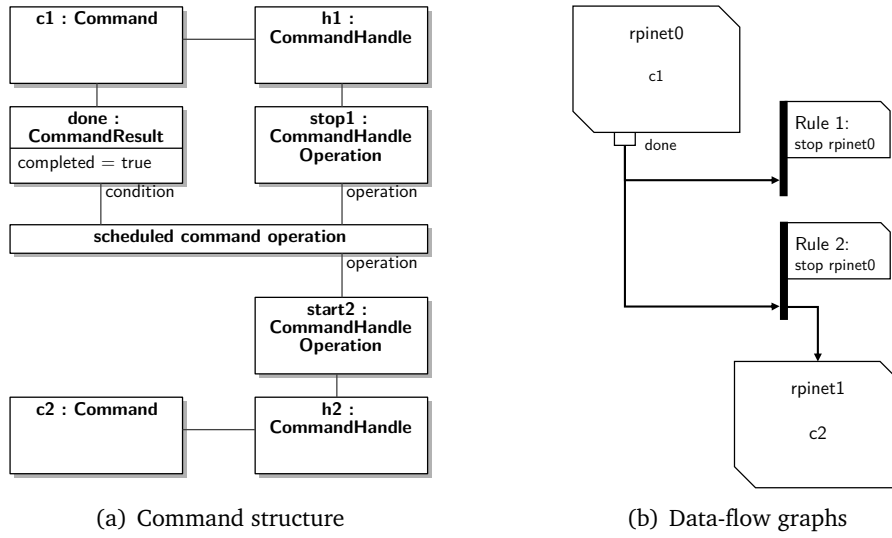


Figure 6.12: Handling Command life-cycle and scheduled operation using synchronization rules

to be constant. Similar steps have to be performed for the different types of Assignments, each using a specialized *RealtimeValueConsumerFragment* to process the value (e.g. forward to the application or change a parameter). To correctly handle the defined *CommandResults* of the Command, for each *CommandResult* a *BooleanNetcommOut* primitive is added to make the value available outside the data flow graph. It is connected to a *RealtimeValueFragment* for the *RealtimeValue* given as a trigger for the *CommandResult*, e.g. to the conjunction of Action and Actuator completion for the Command completion result.

Figure 6.11 gives an example how a simple Command with Action and Actuator is transformed into a data-flow graph. First, the Action is mapped into an *ActionResult* using the *MapperRegistry* and an *ActionMapper*, and afterwards the *ActionResult* is processed by an *ActuatorDriverMapper* to construct a *RealtimeValueConsumerFragment*. This Fragment is added to a Fragment representing the Command, resolving the dependencies to used *RealtimeValues*, and then loaded to the Robot Control Core yielding a *CommandHandle*.

### 6.2.3 Handling Life-Cycle and Scheduled Operations

The Data-flow graph created for the Command can be sent to the execution environment, where it receives a unique identifier that is stored in the *CommandHandle* and can be used to further reference it in life-cycle operations. To handle the life-cycle of the Command, a set of Synchronization rules is issued in addition to the Data-flow graph: For each *CommandResult* defined as completion or error, a Synchronization rule is applied that stops the Command once the *CommandResult*'s *BooleanNetcommOut* becomes *true*. Additionally, the execution environment is requested to report the values of all *NetcommOut* Primitives such as *DoubleNetcommOut*

and *BooleanNetcommOut*, so that they can be forwarded to the corresponding listeners in the application.

To *start()* the Command referenced by a *CommandHandle*, a Synchronization rule with a *true* condition is used, together with the request to start the corresponding Data-flow graph. Similarly, *cancel()* and *abort()* requests are executed through Synchronization rules that reference the Data-flow graph in the cancel or abort list. Scheduled Command operations (cf. section 6.1.3) are also transformed into Synchronization rules: For the given set of *CommandResults*, a conjunction of the corresponding *BooleanNetcommOuts* is created. Additionally, for all started, canceled and stopped *CommandHandles* the corresponding Data-flow graph is added to the start, cancel and abort list. The resulting Synchronization rule is then sent to the execution environment. These Synchronization rules are also assigned a unique identifier, so that their activation can be reported and the application can react correspondingly.

In figure 6.12(a), the two Commands *c1* and *c2* are shown, along with their *CommandHandles* *h1* and *h2*. The Command *c1* has a completion *CommandResult* called *done*. Additionally, a scheduled command operation is given to replace *c1* with *c2* when *done* is reached. In figure 6.12(b), the resulting data flow graphs *rpinet0* for *c1* and *rpinet1* for *c2* are shown. *Rule 1* belongs to the completion result, stopping *c1* once it is *done*, while *Rule 2* represents the scheduled command operation, stopping *c1* and starting *c2* in the same situation. Actually, *Rule 1* is no longer needed when *Rule 2* is defined, however belonging solely to *c1*, *Rule 1* has been created at a time before the scheduled command operation for *Rule 2* was known.

### 6.3 Comparison with Previous Work on the Robotics API

Compared to the work of Angerer [2], various aspects of Command specification and execution were adapted. As mentioned in section 2.1.2, time-variable data was previously separated into *States* and *Sensors*, with *States* representing situations within a Command (i.e. Boolean values that have a scope limited to a certain Command), while *Sensors* were independent from a Command and could be used in multiple contexts. Both *Sensors* and *States* had independent composition operations, while Boolean *Sensor* data could be converted into *States* to allow combination with further *States*.

Apart from duplicating code for *Sensors* and *States*, this method limited the use of *States* to Boolean values and did not allow to define numeric values that are only valid within a given Command (such as the number expressing the progress of a motion, as used now to compute the set-point of an Action). To improve this situation, *States* and *Sensors* were combined into the common concept of *RealtimeValues* (that are optionally limited to a given Command), avoiding duplicate code for composition and transformation to Data-flow graphs, extending the possible range of use and facilitating the way of Action mapping proposed in this thesis (cf. section 6.2.1). For example, using an Assignment with a value based on the progress of a motion allows to

synchronize the stiffness of the youBot arm with its motion progress, an approach suggested for variable stiffness actuators to minimize impact forces when the robot hits an obstacle [99].

Within Commands, for similar reasons the concepts of *SensorListener* and *EventHandler* were combined into *RealtimeValueListeners* that can handle arbitrary *RealtimeValues*. Further notable modifications include the introduction of *Assignments*, allowing to change parameters (such as the stiffness of an impedance controller) in real-time and synchronized to a defined motion, and the introduction of distinguished *CommandResults*. Previously, when a *Command* could have multiple different success outcomes, all could be defined as completion conditions through *CommandStoppers* or as takeover conditions, however a following *Command* could only find out which condition had been taken by inspecting the state of the *Actuators* or environment. The explicit introduction of *CommandResults* makes this decision easier, and provides a consistent way to handle different outcomes with independent successor *Commands*. This is especially important for mobile robots in less structured environments, where the variations possible in the surroundings can lead to different task outcomes that have to be handled separately. Additionally, working with mobility more often leads to motions where the *Command* cannot end when the goal has been reached, but has to further follow the goal to maintain its result state. This happens with arm motions defined relative to the environment (where the arm has to further compensate for platform motions), allowing successors to takeover and continue from this situation, independent from whether they are issued before or after the goal is reached for the first time.

Using the work of Angerer [2], multiple *Commands* could be combined into a *TransactionCommand* that allowed *Command* transitions in real-time. This concept has been dropped in favor of scheduled operations directly allowing to link independent *Commands* to each other with real-time guarantees, providing improvements for the transformation to data-flow graphs. Additionally, using independent *Commands* simplifies reaction to different *CommandResults* of the predecessor and allows to incrementally specify the reactions. This facilitates and simplifies the implementation of reactive behavior, allowing to handle the different outcomes of mobile robot tasks with different reactions. In this context, for mobile manipulators the ability to transition from independent *Commands* for the individual actuators (e.g. bringing arm and platform into suitable start positions) to a task where they are coordinated (running a motion that is achieved through the interplay of platform and arm), and back to separate motions (using the platform to navigate further while the arm performs independent work on a grasped object) offers new possibilities not possible using the previous approach. By default, however, these scheduled *Command* structures do not abort parallel *Commands* when an error occurs within one *Command*; if required this behavior has to be specified explicitly using further scheduled operations. While it would still be possible to implement a similar *TransactionCommand*, this aspect is now rather seen as a responsibility of the *Activity* layer (cf. section 7.3) when handling composition.

Compared to previous work [89, 103], the transformation step of Commands and the resulting data-flow graphs have become easier and less error-prone, whereas the execution environment takes more responsibility for life-cycle handling. Now, the data-flow graphs no longer have to include decision logic when to start, cancel or stop Commands (as these aspects are defined through synchronization rules), and can thus concentrate on the computations required to fulfill their task. Correspondingly, the composition of multiple Commands into one real-time context is now handled as multiple data-flow graphs linked with synchronization rules, deviating from the idea of representing one real-time transaction as a single data-flow graph, while providing the same real-time guarantees as available before. This leads to smaller data-flow graphs that no longer have large disabled regions representing parts that are not active at the moment, and allows better scalability especially on multi-core hardware with poor single-thread performance: Parallel tasks defined to be executed in a synchronized manner can now be evaluated concurrently in multiple threads, instead of being serialized as one data-flow graph. Additionally, this change can be seen as a prerequisite to a distributed implementation of the Realtime Primitives Interface, where hardware devices controlled by a certain Robot Control Core can be handled by data-flow graphs talking only about the corresponding device (avoiding the distribution of a single data-flow graph onto multiple computers), and real-time coordination only has to be performed on the level of synchronization rules.

One further change in the transformation process is the handling of Actions. While previously directly transformed into a data-flow graph, Actions are now first transformed into a `RealtimeValue` representation, allowing to use real-time value definition and calculation methods and significantly reducing the code size of many ActionMappers. Additionally, most parts of an Actuator now result in parts semantically similar to Assignments, describing a `RealtimeValue-ConsumerFragment` that handles `RealtimeValues`. `RealtimeValueFragments` no longer contain the primitives representing their dependent `RealtimeValues`, but rather add semantic dependencies describing that other `RealtimeValues` should be linked to the corresponding Fragment. This way, the structure of data-flow graphs becomes clearer, and more optimizations are possible to avoid duplicate data-flow graph parts by reusing `RealtimeValues`, avoiding to instantiate Primitives that are later removed by the automatic net optimization step used before execution [103].

Additionally, data type conversions are now also handled explicitly on `RealtimeValues`, instead of implicit *converters* [89] (also called *link builders* by Vistein [103]). These *converters* were used to access data of a certain type (e.g. the joint angles of a robot, or the position of a certain Frame relative to another), and to convert data into other formats (e.g. from a goal to a position, from Joint space to Cartesian space, or from one reference Frame to another). This conversion was based on the data types (e.g. being a transformation expressed in a certain reference Frame) but no further semantics (e.g. being the current or the desired position). Due to their large feature set, common implementation pitfalls were to trigger the wrong conversion, e.g. receiving the current robot position instead of the desired set-point when no suitable link builder was found

to convert the given set-point into the desired goal data format. The new, more explicit version based on *RealtimeValues* avoids this type of error, and additionally makes conversion choices more explicit (e.g. whether to convert a Cartesian goal into a Joint goal first and then into a Joint position, or to a Cartesian position first and then into a Joint position). In this context, the intermediate composition of Primitive ports to data-flow ports [103] has been omitted by placing semantics into the data types (e.g. Pose and Velocity) and *ActionResults* (providing goal positions and velocities through different methods), further reducing the complexity and number of indirections used in the transformation process.

## 6.4 Related Work

The way of command specification and execution introduced in this chapter (adapted from Angerer [2], Schierl et al. [89], and Vistein [103]) differs from the way other robot frameworks use. Extending the comparison with related work presented in the aforementioned references, this section goes into detail about two approaches that are especially popular in mobile robotics.

In *ROS* [82] as a typical component framework, task execution occurs through the interplay of different components. Tasks are commanded through *Messages*, *Service* invocations or *Actions*. *Messages* are sent without acknowledgment, and are thus used when no feedback about task execution or completion is required. *Services* allow remote procedure calls that provide a reply once the task has been executed, however do not allow to interrupt the task. *Actions* are handled through the *actionlib* framework [28]. It allows to start long-running, preemptible tasks that provide feedback information and notify about their result, but may also be canceled if a goal has to be changed. *Messages*, *Services* and *Actions* are handled by components that process the corresponding requests and pass further data to other components. For example, a *joint\_trajectory\_controller* accepts a *JointTrajectory Message* or *FollowJointTrajectory Action* describing (spline) set-points for a joint motion, and cyclically forwards positions, velocities or efforts to a *hardware\_interface* component. Additionally, it is possible to update the active trajectory, replacing all trajectory points that are given for future time instants. Updating trajectory points thus leads to a motion that depends on the exact moment when the update is sent.

However, *Messages*, *Services* and *Actions* are transferred over a network and do not provide real-time guarantees, thus failing at design goals 6.1 and 6.2. To support further motion types, new controller components have to be implemented that accept new *Messages* or *Actions* describing the motion. When multiple motions are to be executed as a sequence, a coordinating component invokes multiple *Actions* one after another, specified in (typically C++ or Python) code or through a *SMACH* state machine (cf. section 7.5). However, this specification method does not provide timing guarantees and thus cannot provide the precision possible with the proposed approach.

Using *OROCOS* [14], the typical approach before version 2 consisted of a trajectory interpolation component from the *motion\_control* package providing set-points for a given motion, along with a component from the *hardware* package that handles the set-points and forwards them to the real hardware devices. The trajectory generators however did not support pre-emption, as this was seen as a coordination aspect that should be handled outside trajectory computation. Therefore, further trajectory generators were required that could stop the robot. Based on this design decision, braking the robot in a trajectory-preserving manner when an interruption occurs is hard to achieve and requires to change the component connection in real-time: The trajectory interpolation component has to be disconnected from the hardware component, inserting an intermediate component that uses the received trajectory to plan the braking trajectory and forward it to the hardware. In contrast, in the proposed approach trajectory-preserving brakes are a part of the Action implementation, avoiding this complexity.

To start or sequence different motions, the required components were instantiated and connected through a deployment configuration or script, and started or stopped accordingly. For more complex interactions, state-machine descriptions using rFSM (introduced by Klotzbucher et al. [58]) could be used. In this context, Bruyninckx et al. [15] suggested separating concerns in robot programming according to the 5C:

**Computation** aspects contain calculations that are to be performed on data (such as control loops or trajectory planning), providing the main functionality of the robot system.

**Communication** between different components (and also between different computers) should be made explicit and separated from computation, to make computation components more reusable and handle communication issues in clearly defined spaces.

**Coordination** handles the decision when the system should do what, defining discrete behavior in the form of a finite state machine (that only works on Boolean events and may not perform computations itself).

**Configuration** provides parameters to computation and communication components, adapting them to current needs (to be kept outside other components to make them reusable in other settings where a different configuration is required).

**Composition** decides how the different components interact with each other, thus changing the topology of used components according to the current task.

This separation of concerns also influenced some aspects of the proposed framework that changed in comparison to previous work: Action implementations are now handled as pure computation descriptions that are no longer directly mapped to data-flow graphs, but are described as *RealtimeValue* calculations that can be transformed into different executable forms. Combining multiple Commands into real-time transactions as a typical coordination aspect now

leads to separate data-flow graphs (representing the computations for each Command) linked through Synchronization rules (representing the coordination aspect), instead of a single data-flow graph encoding the switching logic and all Commands. The use of DeviceParameters allows to handle configuration aspects, separating them from computation and communication. The composition aspect describing the links between different data-flow graphs and their connection to device drivers is defined from within the data-flow graphs, and realized by the framework. In contrast to *ROS*, communication between different computer systems is handled explicitly (as described in chapter 9), while the communication with hardware devices is managed through device drivers (that are independent from Commands or data-flow graphs), thereby separating the communication concern.

In version 2 of the *OROCOS Toolchain*, hardware and application specific components have been removed. Thus, the trajectory interpolation and hardware components are no longer provided, while the *OROCOS Toolchain* focuses on being a pure framework for application-specific real-time control. It finds its role as a way to implement *ROS* nodes providing capabilities that offer real-time guarantees within the execution, and is furthermore used in the context of domain-specific languages and generic components that more closely follow the *5C* approach. For example, the *iTaSC* implementation introduced by Vanthienen et al. [101] (further detailed in section 5.4) or the use of state machines in *rFSM* [58] (further detailed in section 7.5) are examples of this development. Additionally, the reference implementation for this thesis is partly based on the features provided by the *OROCOS Toolchain*.





## DEVICE CAPABILITIES AND SYNCHRONIZATION

Typical robot applications do not consist of one single action performed, but of a set of different capabilities applied sequentially or in parallel. Additionally, multiple devices or robots can be used, demanding synchronization. Revisiting Requirement 4 requesting task composition and continuous motions spanning multiple tasks, further design goals were derived concerning device capabilities and synchronization: To make applications reusable for different devices and robots, the Devices should share common interfaces to be used in applications.

**Design Goal 7.1.** *Provide access to device capabilities in a device-independent way.*

For example, motions in Cartesian space should equally be available to industrial robot arms, but also mobile platforms and even quadcopters. Additionally, especially in mobile and service robotics, Devices are not limited to a predefined set of capabilities (which has been a viable approach in industrial robotics for many years), but it should be possible to implement and add further extensions to existing Devices to fulfill new needs to be addressed by Devices.

**Design Goal 7.2.** *Support an extensible set of device capabilities.*

Having a direct effect on the physical world, device capabilities often require considerable execution time that is not bound by CPU performance. When applications include planning steps or other long-running tasks aside such device capabilities, it should be possible to perform the side tasks in parallel with the capabilities instead of waiting for completion, making use of the processing resources available and improving overall execution duration. This becomes especially important with mobile robots, where variations in the environment do not allow to always use the same motion, but require dynamic planning based on the current environment state.

**Design Goal 7.3.** *Decouple application workflow and capability execution to avoid unnecessary dead-times.*

This requirement is also the basis for one way to achieve multiple motions without stopping the robot in between, as requested by Requirement 4, because the following motions can be prepared while the first motion is executed and can thus be appended while the robot is still in motion. Another way is to directly combine the motions into a single capability, requesting for capability composition:

**Design Goal 7.4.** *Support the combination of (basic or composed) device capabilities according to common coordination patterns.*

This can be used for the interplay of platform and arm of a mobile manipulator, defining a common behavior through the interplay of basic capabilities. Apart from sequential and parallel composition, mobile robot systems in uncertain or unstructured environments often have to react to sensor data or further events. In this context, it should be possible to define reactive behavior.

**Design Goal 7.5.** *Allow the definition of capabilities with reactive behavior based on (basic or composed) device capabilities, supporting repetitions and case distinctions.*

Once multiple devices are used at the same time, the matter of synchronization occurs: It becomes important to decide when to execute which capability relative to other devices and events.

**Design Goal 7.6.** *Allow to synchronize the execution of capabilities of different devices.*

In section 7.1, different types of synchronization are introduced. As a basis of synchronization and composition, the individual capabilities are modeled as Activities, as described in section 7.2. These Activities can be used to describe a single capability, but also be combined towards more complex workflows (section 7.3). These concepts are based on the work of Angerer et al. [3], but are adapted to fulfill the new design goals. Section 7.4 gives an overview about the modifications and extension applied (in comparison to [2]), and how the modifications relate to service modeling introduced by Hoffmann [39]. Finally, section 7.5 gives an overview of other approaches to model tasks and their composition and synchronization.

## 7.1 Levels of Robot Synchronization

When multiple robots are to work together, a certain amount of coordination and synchronization becomes important to reach the desired goal. Depending on the situation, different types of synchronization posing different requirements can be used. This work distinguishes between three types of synchronization:

**Workflow synchronization** is used to synchronize the workflows and high-level tasks performed by the different robots. This type of synchronization is often called “coordination” (cf. [58, 83]) and defines constraints such as that the task of one device has to be completed successfully before a task of another device may start.

**Time synchronization** can be required for concurrent task execution. The goal here is to make sure that two tasks of different devices are started exactly at the same time, and optionally that the execution duration of the tasks is synchronized.

**Data synchronization** means that latest information about one device is available to another device at run time, ideally with defined time-constraints. This can include geometric data, but also sensor values.

These types of synchronization can be achieved in different ways. The following sections describe workflow synchronization (cf. section 7.1.1), time synchronization (cf. section 7.1.2) and data synchronization (cf. section 7.1.3), and how they can be achieved through communication or observation.

### 7.1.1 Workflow Synchronization

The first type of synchronization is Workflow synchronization, coordinating the tasks executed by different devices. The mechanisms available depend on the application structure. Within a single workflow, the order of execution for multiple tasks is given as a sequence. This sequence can be expressed through a sequential program, through a list of tasks to execute sequentially, or through reactive behavior specified as a state chart (cf. section 7.3). However, the workflow in this scheme contains no implicit parallelism, i.e. the parallel execution of two actions has to be explicitly specified as one task. This structure has the advantage of defining the workflow in one place, making it easier to understand. However, there is also a single point of failure, and the one application must be allowed to access and control all devices.

More flexibility can be gained using multiple workflows: Here, a separate workflow is defined for different device groups, or even for each single device. In order to synchronize these individual workflows, different mechanisms can be used: If the workflows are executed in different threads of the same application, programming language constructs for synchronization can be used, e.g. *Mutexes* or *Semaphores* allowing to coordinate the different threads. Alternatively, communication is used: In state charts, one workflow state chart can signal an event, triggering transitions in other state charts. Additionally, shared variables can be used to signal events or to convey state from one thread to another. When using multiple applications (cf. section 2.3.3), network or inter-process communication are available to signal events or request action from other robots. A third method is through observation: One workflow can use sensor data or observations to determine the current situation and find the right time to act. This is especially helpful when explicit communication between the different applications

is not possible or not allowed. Based on the observed state, decisions are taken and tasks are executed. Typical observations include triggers defined on the world model, e.g. reacting once a relevant position change is seen.

Generally speaking, Workflow synchronization does not require special care on the execution environment, because all coordination can be handled at application level without real-time guarantees.

### **7.1.2 Time Synchronization**

The next synchronization type with higher timing requirements is Time synchronization. Here, the goal is to start two tasks exactly at the same time, or with a specified time offset. Within one execution environment, this kind of synchronization is easily possible: In the proposed software framework, Synchronization rules allow to specify that two Commands should be started at the same time. If different execution environments are to be synchronized, more work has to be done. The easiest way to solve this assumes that a signal can be sent that is received by all execution environments at the same time (within the allowed timing inaccuracy). In this case, on each execution environment a sequence can be started that first waits for the signal, and afterwards runs the desired task. The application can then send the signal to start the tasks in a coordinated fashion. Other mechanisms include using a synchronized clock to specify the start time for the tasks, but require bounded communication delays to make sure the required data is transmitted before the specified start time.

Time synchronization can be required when two tasks are planned together, exploiting knowledge about each other's progress. For example, using two industrial robot arms to cooperatively carry a work piece can be solved through time synchronization [2, chapter 10.6]: After planning a trajectory for the work piece, both manipulators can be commanded to follow the trajectory using an appropriate motion center. As the two motions are started exactly at the same time and follow the same work piece trajectory, the resulting motion of the robots is appropriate to move the work piece along the defined trajectory. Further examples include collision-free motions of multiple robots with intersecting work envelope, as well as tool actions that are to be executed at a certain position of a motion. However, for realizing load sharing of multiple mobile robots this method solely relying on Time synchronization is not sufficient, because uncertainties in the positions of mobile robots cannot be corrected.

### **7.1.3 Data Synchronization**

The strongest form of synchronization is Data synchronization. Here, live data from other devices can directly be accessed at run time. This allows to incorporate other devices' behavior and state into the task execution. Multiple techniques can be used to achieve Data synchronization. Assuming that a reliable, high-performance communication channel exists, the corresponding data can directly be transferred and made available. This is especially possible if the devices are

connected to the same computer and reside in the same execution environment, where real-time communication is possible. But also wired network connections fulfill these requirements. When the required data is visible and can be observed, for example geometric or position information, another option becomes possible: The data can be measured using a sensor available in the destination execution environment, thus providing the current data in real-time.

Having access to foreign data further increases the possibility of physical cooperation: Motions can be specified relative to other devices' positions, thereby becoming independent from their concrete behavior. As an example, one industrial robot may work on a work piece that has been grasped by a second robot (cf. [2, chapter 3.1]). Through data synchronization, it is enough to define the work performed by the first robot relative to the work piece. During execution, the current position of the work piece can be retrieved from the second robot, and integrated into motion of the first robot.

## 7.2 Modeling Device Capabilities

As introduced in chapter 6, executable tasks can be defined as Commands containing an Action and an Actuator. To simplify programming and allow more advanced modeling concepts, an additional layer is added on top of Commands, introducing the concept of Activitys. An **Activity** encapsulates an Actuator capability – composed of one or multiple basic tasks – that adapts to the result of its predecessors and provides metadata about its possible results. For execution, Activitys rely on Commands and Synchronization rules created specifically for the given situation, provide information about the state of the world after execution, and exhibit special planning and execution semantics.

Application programmers can access features of Devices through DeviceInterfaces. These **DeviceInterfaces** are linked to Devices through composition, allowing to access an extensible set of capabilities. Using the **use()** method of Devices (cf. figure 7.1), an instance of a specified DeviceInterface can be retrieved. For Sensors, **SensorInterfaces** provide access to sensor data, while for Actuators specialized **ActuatorInterfaces** are used to model capabilities. ActuatorInterfaces serve as Activity factories that can be used to create Activitys for a given task.

### 7.2.1 DeviceInterfaces and Activities – Accessing Device Capabilities

For basic motions, these factories are grouped similar to the Actions available for Actuators: In Cartesian space (cf. section 5.1.1), DeviceInterfaces are available for process as well as goal motions, both in position and velocity space. Figure 7.1 gives an overview over the corresponding interfaces, including **CartesianPathMotionInterface** and **CartesianGoalMotionInterface** in position space and **CartesianVelocityMotionInterface** in velocity space. These interfaces allow to obtain Activitys for the corresponding capabilities. Activitys know the Devices they work on, and provide four methods concerning execution:

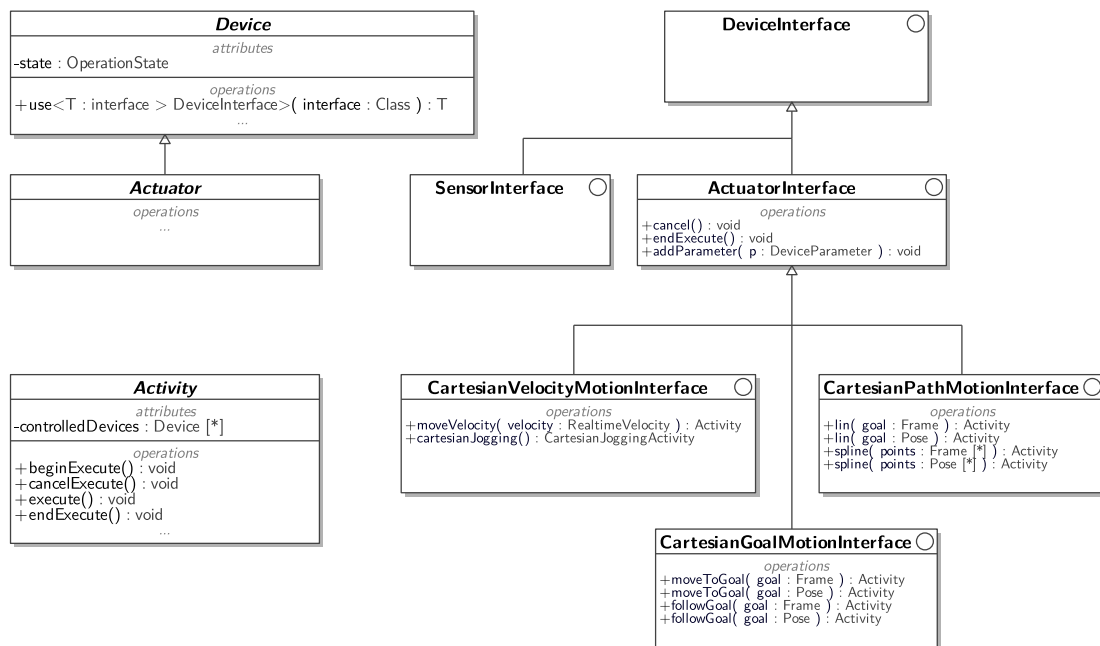


Figure 7.1: Class diagram of concepts related to Activities and DeviceInterfaces

**beginExecute()** triggers the start of an Activity, and returns the control flow when the Activity execution has started. If another Activity has been executed on the same Device before, this will happen when the previous Activity has reached a state the new Activity can cope with.

**endExecute()** waits for a started Activity to finish, i.e. reach a state that is marked as complete. Looking at the underlying Command layer, this does not necessarily mean that all used Commands have ended; some may still be running to maintain the achieved state.

**execute()** starts an Activity and blocks the control flow until the Activity is completed. It can be seen as the sequential execution of **beginExecute()** and **endExecute()**.

**cancelExecute()** signals an Activity that it should come to an end, similar to cancel requests for Commands.

For purely sequential workflows, it is possible to execute multiple Activities in a sequence using **execute()**. However, then the code executed between the Activities and also the preparation of Activities happens after the previous Activity has completed, leading to a noticeable delay between the Activities. To improve this situation, **beginExecute()** can be used. Here, the control flow continues once the Activity has started, allowing to prepare the next tasks while the Activity is still running. This way, the perceived time delay is reduced, and additionally it becomes possible to use successor Activities that do not have to wait for the previous Activity to complete, but can take over at an appropriate time during execution. This concept – called **Takeover** –

can occur at certain times during Activity execution, when a so-called takeover condition is active and the following Activity can cope with the situation described by this condition. For example, this is used in conjunction with motions in industrial robotics for a concept called *motion blending*: Quite often, the direct path between two points is blocked, however another path that uses some intermediate points can be found. In this case, using `execute()` for the motions to the intermediate points causes the robot to stop at each of these points. However, at such intermediate points it is not required to really stop the robot, so the path can be “blended” towards the next motion once the robot is sufficiently close to the intermediate point. This behavior can be achieved using `beginExecute()`: In addition to the goal position, the motion to the intermediate point defines a takeover condition where the path may be left. As the next motion is already being prepared while the first motion is still executed, it can plan a motion starting at the takeover place (e.g. using `JointTrajectoryFromMotion` as introduced in section 5.1.2) and thus continue the robot without stopping at the intermediate point. Further uses include preempting a running Activity when a predefined situation occurs (e.g. an error or a certain sensor measurement), and continuing with another one.

Using `beginExecute()` however only provides a best-effort semantics: If it is called too late, a takeover point of the previous Activity can already have passed, and the previous Activity thus has to be executed to completion before the new Activity can start. Additionally, `beginExecute()` has one drawback concerning error handling: If an error occurs while executing the Activity, control flow has already continued and cannot immediately cause an exception. The corresponding exception is thus encapsulated into a **PreviousActivityExecutionFailedException** and thrown when the next Activity for the same Device is started. Applications have to be aware of this behavior to make sure the right reaction to the error is applied.

Figure 7.2 shows an example workflow with the four Activities *a*, *b*, *c* and *d*. After *a* has been started using `beginExecute()`, some internal work can be performed. The following call to `beginExecute()` for *b* blocks until *a* is completed, and starts *b*. In this example, an error occurs at the execution of *b*. However, the control flow is within the internal work of the application, so the Exception cannot be thrown immediately. Instead, it is thrown for the call of `beginExecute()` for *c*. After the exception has been thrown, following Activities (such as *d*) can be started normally.

Semantically, an Activity encapsulates a device task, and thus has to provide real-time guarantees if required for the specific task. From this point of view, three types of Activities exist: Some Activities are *deterministic*, guaranteeing that each execution in the same context will behave exactly alike, with the same duration and the same possible outcomes (known in advance). For example, a `JointTrajectory` executed by a robot arm falls into this category. Other Activities are not totally deterministic, but *pre-plannable* in the sense that the possible outcomes are known in advance, but execution time can vary. As an example, moving to a Cartesian goal determined by a sensor cannot provide its execution time in advance, but initially knows that

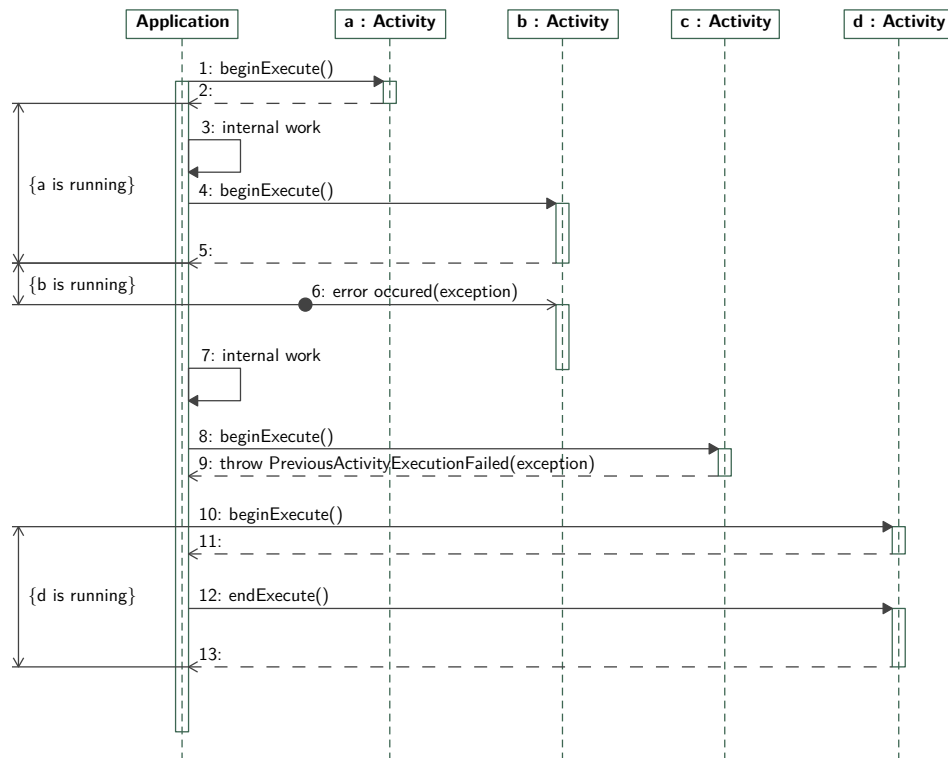


Figure 7.2: Parallelism and error handling with `beginExecute()`

the robot will be at the defined goal after execution (if no error occurs). Other Activitys neither provide deterministic execution, nor can describe their outcomes beforehand. There, the exact behavior and the resulting outcomes only become known during run time, e.g. for a grasping Activity that determines the best grasp strategy for an object based on sensor data perceived during the grasp process. Still, all three types of Activitys can be seen as good abstractions for their described task.

### 7.2.2 Implementing and Executing Activities through Commands

When comparing Activities to Actions, the LIN Action requires a start and goal Pose, while for the *lin()* method of the CartesianPathMotionInterface the goal Pose is sufficient. This becomes possible because Activities – as opposed to Actions or Commands – store metadata about their expected outcomes that can be used by following Activities.

Figure 7.3 gives an overview of the concepts required to offer this functionality. An Activity has a set of possible **ActivityResult**s describing situations that can occur during execution of an Activity. Each ActivityResult contains an extensible set of metadata in the form of **Property**s, e.g. describing the current Pose where a robot is when the ActivityResult occurs, or the goal where the robot has been commanded to go to. Additionally, the ActivityResult contains two sets of



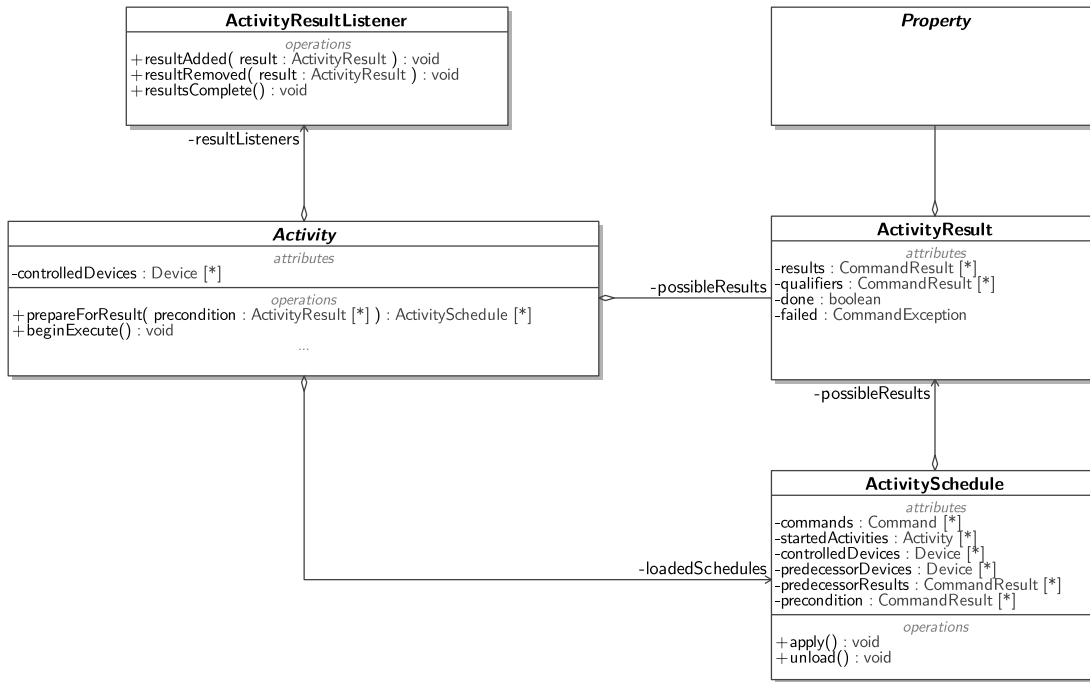
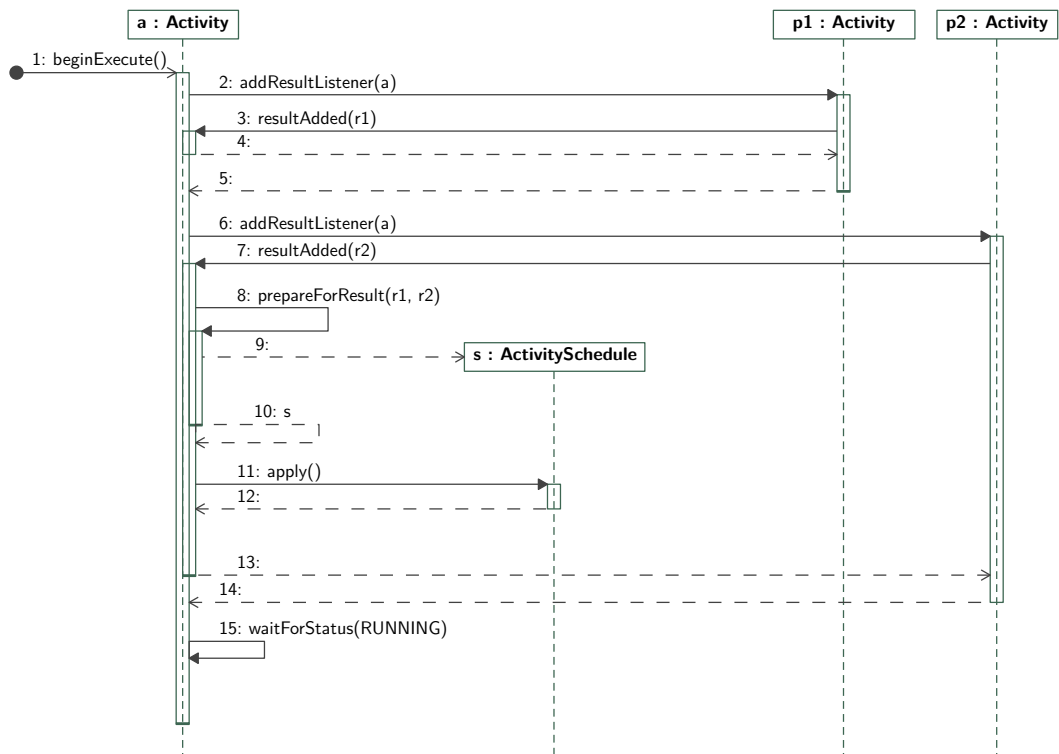


Figure 7.3: Class diagram with concepts related to Activity execution

CommandResults called *results* and *qualifiers*. The *results* describe the state of the Commands used to implement the capability, while the *qualifiers* describe additional conditions that have to be true for this **ActivityResult** to happen, talking about other Commands that do not belong to the main capability. Furthermore, each **ActivityResult** can have the semantics of being **done** or **failed**. In this context, done has to be seen as logical completion, which does not require the corresponding **CommandResult** to be completed. An Activity that moves the robot to a goal can be done, although a Command is still running that tries to follow the goal if it moves away.

Based on the **ActivityResult**s, the follow-up Activity can decide if and how it can perform its work. Therefore, the operation **prepareForResult()** has to be implemented. It is called for each post condition of the predecessor(s), and decides how to react in this case. The post condition consists of a set of **ActivityResult**s belonging to the previously executed Activities, so that each Device used in the Activity is also present in one **ActivityResult**. The preparation returns a set of **ActivitySchedules** that describe how to handle the given **ActivityResult**s.

The **ActivitySchedule** describes which Commands have to be executed, the predecessor results and further preconditions for its execution, and the set of Activities that are to be seen as started once this schedule is executed. Additionally, it describes which Devices are controlled, and which Devices were controlled by the predecessor(s). The latter information is required to decide before execution if the **ActivitySchedule** is complete in the sense that all predecessor Devices will be handled, while still allowing to return incomplete **ActivitySchedules** that can be combined in


 Figure 7.4: Preparation process for `beginExecute()` with an Activity covering two Devices

composed *Activitis*<sup>1</sup> to create a complete *ActivitySchedule*.

An *ActivitySchedule* can be applied, loading all the contained *Commands* on the corresponding execution environment, and scheduling the start of the *Commands* when the given *CommandResults* occur. Applied *ActivitySchedules* contribute to the possible *ActivityResults* provided by the *Activity*. *ActivitySchedules* can also be unloaded if they are no longer required because the precondition did not occur.

**ActivityResultListeners** for an *Activity* are informed about new *ActivityResults* that become possible, as well as about *ActivityResults* that can no longer happen. Additionally, they are notified when the set of *ActivityResults* is complete in the sense that no further *ActivityResults* will be added, but only existing ones that become infeasible will be removed.

When an *Activity* is to be started, first its *Devices* are inspected. For each *Device*, the *Activity* previously executed is looked up, and a result listener is added. From all the results, the cross product is created, and forwarded to `prepareForResult()`. If the method provides valid schedules, they are applied. All these steps are executed in an event-based manner, to allow *Activitis* to provide their results incrementally or at a later stage. Figure 7.4 shows an example sequence that can happen for `beginExecute()` for *Activity a* that covers *Devices* previously controlled by *Activitis p1* and *p2*. For both predecessors, *a* adds an *ActivityResultListener* that is informed about the

<sup>1</sup>More details on the combination of *Activitis* can be found in the following section.

results  $r1$  and  $r2$  respectively. Once  $r1$  and  $r2$  have arrived, both are passed to `prepareForResult()`. This method plans the Activity execution for the given situation, and returns an `ActivitySchedule`  $s$  that is subsequently loaded. To achieve its expected semantics, `beginExecute()` then waits until the Activity execution starts.

With respect to `ActivitySchedules` and `ActivityResults`, `Activitys` can be differentiated into two types: Some `Activitys` will provide all their possible results immediately, especially before execution. These `Activitys` can be combined with real-time guarantees, because it is possible to prepare the following `Activitys` based on the results before the first Activity is even started (cf. section 7.3.1). Other `Activitys` only provide some of their possible results later at run time, in which case the preparation of the following Activity can be too late. However, these `Activitys` allow to handle sequences incrementally without having to plan the entire job up front, and especially allow the specification of unbounded loops. As a drawback, these `Activitys` only provide real-time guarantees within their children, while the transition between `Activitys` is performed in a best-effort fashion similar to using `beginExecute()`.

Being fully implemented on top of `Commands`, the execution of `Activitys` poses no further requirements to the execution environment. In their implementation, `Activitys` use scheduled `Command` operations to handle the transition between different tasks, as well as for different types of composition available on the Activity level. Notifications about Synchronization rule activation are used to monitor progress, e.g. to find out when `Activitys` are started or completed.

## 7.3 Composed Activities – Combining Capabilities

If multiple `Actuator` capabilities are to be applied one after another in an application, and no exact timing is required between the capabilities, the easiest way is to model this sequence in the control flow of the application programming language. Figure 7.5 gives a logical view of such a control flow, abstracting from Activity execution internals (for clarity of presentation, `ptp()` is here shown as a method of `Device` instead of the corresponding `DeviceInterface`). The execution of `ptp(b)` blocks until the previous motion `ptp(a)` has completed, however internal work can be executed between issuing the motion tasks.

In control flow, the corresponding `Activitys` can be executed using `execute()` or `beginExecute()`, while any further computations belonging to the application logic can be performed between starting the `Activitys`. This method also allows for loops and case distinctions in the control flow, however in the case of `beginExecute()` code with its control structures is executed before the corresponding Activity has completed, which can lead to incorrect decisions.

When working with multiple `Actuators`, a certain amount of parallelism can be achieved through `beginExecute()` (cf. figure 7.6): If `Activitys` for different `Actuators` are executed, for each Activity the control flow will be blocked until the previous Activity concerning the same `Actuator` is completed, while allowing `Activitys` of other `Actuators` to continue in parallel (cf. `ptp(a1)` and

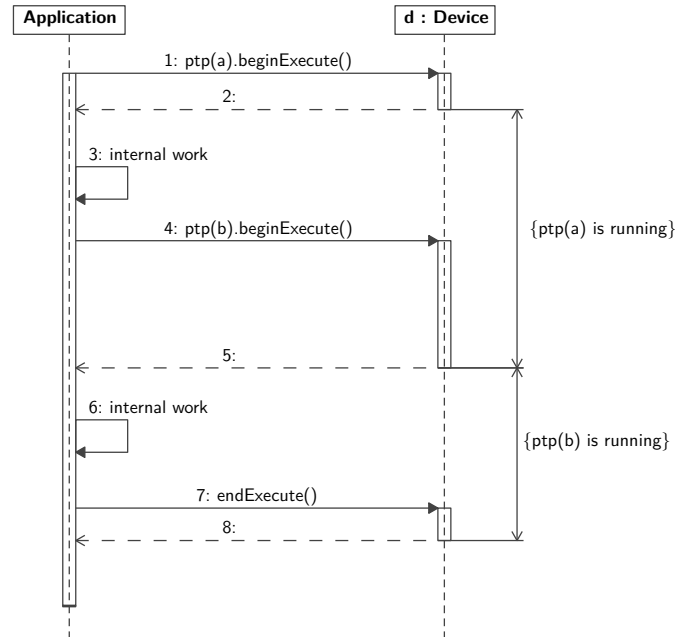


Figure 7.5: Control flow of an application using `beginExecute()` for sequential execution

`ptp(a2)`). However, `beginExecute()` still blocks, so an Activity for a second Actuator that will be started as a next step can only start once the first Activity has been started (leading to the delay before `ptp(a3)`, because `ptp(b2)` blocks). This method provides the easiest form of workflow synchronization, where synchronization points can be achieved through the use of `execute()` or `endExecute()`. To achieve more parallelism, further concepts are required as detailed in the following sections.

### 7.3.1 Parallel and Sequential – Composition with Real-Time Guarantees

When trying to make a sequence of Activities reusable by providing it through an `ActuatorInterface`, it becomes helpful to encapsulate it into an Activity. One way is to combine a list of Activities into a **SequentialActivity**. Here, the Activities are executed sequentially, and for each Activity it can be defined if it has to complete or may be taken over, e.g. to use motion blending. The resulting Activity is *deterministic* if the inner Activities are, and is *pre-plannable* if all inner Activities are at least *pre-plannable*, because the entire sequence is known beforehand and is completely prepared and loaded before starting. However, if an inner Activity is not *pre-plannable*, the composition as well cannot provide its results beforehand, and can thus perform the Activity transitions only on a best-effort basis.

For this Activity, the entire sequence must be specified at creation time, and cannot be influenced by calculations executed in the application at run time of the Activity. Additionally, as it is completely prepared beforehand, the entire planning has to be performed before the

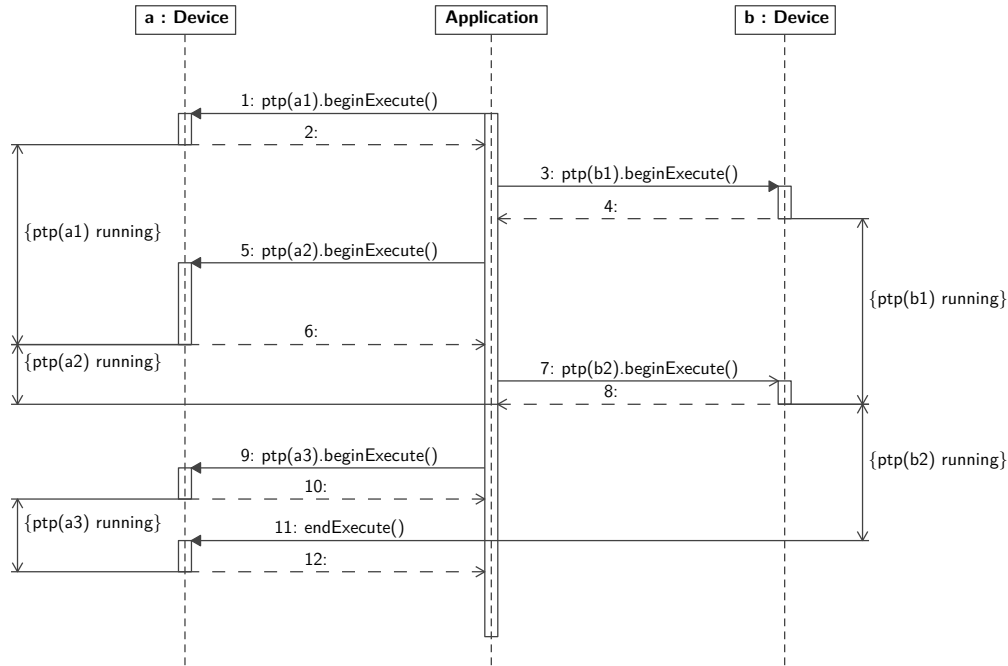


Figure 7.6: Control flow of an application using `beginExecute()` for parallel execution with two Actuators

Activity starts, and thus cannot use the execution time of the early steps for planning the later steps. When the used Activities support multiple possible outcomes (e.g. completion and takeover), planning must be performed for each outcome, causing exponential growth for longer sequences.

Without loss of generality, we will focus on a sequence of two Activities. The implementation of `SequentialActivity` forwards the `ActivityResults` received in `prepareForResult()` to its first inner Activity. The created `ActivitySchedules` are then augmented so that once loaded, they forward their possible results (maybe limited to the ones that are completed) to the second Activity. These resulting `ActivitySchedules` are then also applied, and all `ActivityResults` provided by them are returned as `ActivityResults` of the `SequentialActivity`. When more than two Activities are specified, a `SequentialActivity` can be created that combines the first Activity with the sequence of following Activities, recursively reducing the problem to the two Activity case.

Similarly, a **ParallelActivity** is available that executes two Activities with different controlled Devices in parallel with real-time guarantees. It can be used to explicitly handle parallelism required within workflows. Its implementation forwards the `ActivityResults` to both Activities and combines the resulting `ActivitySchedules`, as long as they use mutually exclusive devices. Completion is signaled once both Activities are completed, and further `ActivityResults` originate from the cross product of results of both Activities. Using a `ParallelActivity` provides time synchronization, guaranteeing that the parallel Activities are started exactly at the same time.

```

1  // Starting a thread activity for the arm ...
2  new ThreadActivity(arm) {
3      @Override
4      protected void run() throws RoboticsException {
5          arm.use(PtpInterface.class).ptp(a).beginExecute();
6          // doing some internal work ...
7          arm.use(PtpInterface.class).ptp(b).beginExecute();
8      }
9  }.beginExecute();
10 // ... doing some internal work ...
11 // ... and appending another motion of the arm ...
12 arm.use(PtpInterface.class).ptp(c).beginExecute();
13 arm.use(PtpInterface.class).endExecute();

```

Figure 7.7: Java code using a ThreadActivity

To allow simple case distinctions evaluated in real-time, a **ConditionalActivity** allows to define a condition choosing between two execution paths given as further Activities. Here, the ActivityResults received in `prepareForResult()` are filtered or augmented by the condition, and forwarded to the corresponding Activity for that case. The resulting ActivitySchedules as well as their possible ActivityResults contribute to the results of the ConditionalActivity. Applying all schedules then leads to the desired behavior. Once the decision has been taken, the set of possible results can be reduced by removing all results from the Activity not taken.

### 7.3.2 ThreadActivity – Encapsulating Long Sequences

In other cases, when encapsulating sequences where real-time guarantees between the combined Activities are not required, a **ThreadActivity** can be used that behaves similar to the control flow case described in section 7.3.

To use it, a set of controlled Devices has to be given and a `run()` method has to be implemented that contains the control flow to be executed in an extra thread when the Activity is started. From the outside view, a ThreadActivity behaves like a regular Activity, supporting to take over the previous Activity (allowing motion blending into its first motion), as well as being taken over by the next Activity (allowing to blend into the following motion if the last Activity is started using `beginExecute()`). When `beginExecute()` is called, it blocks until the first Activity defined in the thread has been started, and `endExecute()` awaits the completion of the thread as well as the last Activity executed in there.

Figure 7.7 shows example code using a ThreadActivity, while its execution trace is given in figure 7.8. There, the ThreadActivity  $t$  along with the run times of  $t$  and the Activities started within  $t$  are shown. The call of `ptp(c).beginExecute()` blocks until  $t$  is completed, not accepting the motion `ptp(c)` to be executed between `ptp(a)` and `ptp(b)`. In this regard, the ThreadActivity differs from using a traditional thread, because it blocks the used Devices and thus makes sure

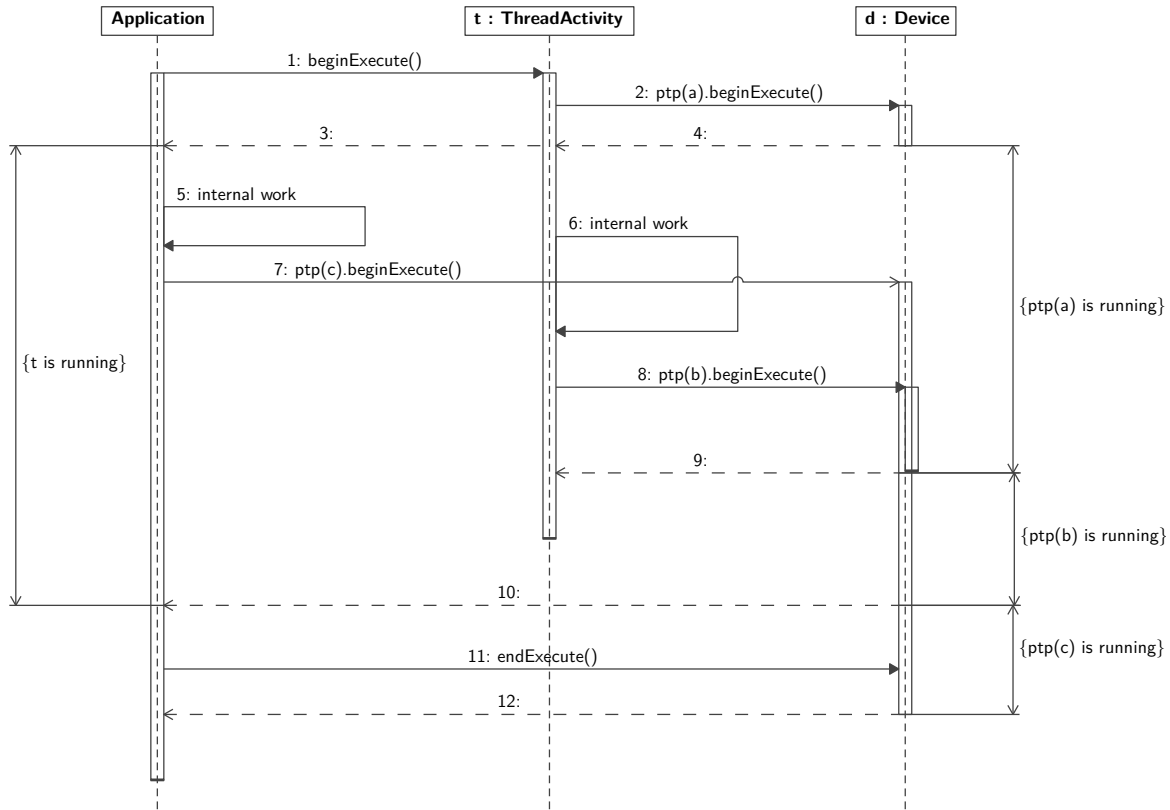


Figure 7.8: Example flow for the code given in figure 7.7

that the internal work of *t* is not interrupted

To implement this behavior, a ThreadActivity manages its own set of predecessor Activities. These are used to determine the ActivityResults that need to be handled when a new Activity is started within the thread. The set of predecessors is initialized with the ActivityResults received in `prepareForResult()`, and updated when an Activity is executed from within the thread. Once the thread is completed, the results from the last executed Activity are returned as ActivityResults of the ThreadActivity.

Within a ThreadActivity, arbitrary control flow can be executed, with the same semantics as in usual control flow working with Activities (`beginExecute()` allows asynchronous execution and will continue, however exceptions will only be thrown at the next Activity), allowing loops and case distinctions in the application code. In this case, case distinctions are decided on at run time, so the combinatorial explosion seen in SequentialActivities does not occur in this case. However, as the Thread is executed in the application in parallel, no real-time guarantees can be given because the control flow or planning of the next Activity may not be completed before the predecessor finishes. Additionally, in ThreadActivities the final set of ActivityResults cannot be given in advance, but only after the thread has completed, emphasizing the need for an incremental reporting of ActivityResults. When executing multiple ThreadActivities in parallel,

workflow synchronization between the different threads can be achieved through standard programming language constructs such as *Mutexes* or *Semaphores*.

### 7.3.3 StateChartActivity – Encapsulating Reactive Behavior

When handling mainly linear problems, maybe with some error cases interrupting the process, using control flow to specify the behavior is an easy and comfortable way. However, if the desired behavior is more dynamic, a pure description as control flow becomes more confusing and complex. This is especially the case in situations where the same behavior has to be applied as a reaction to multiple different situations, and where multiple case distinctions decide about the next step to be taken. These situations often occur with reactive systems that decide their behavior based on (sensor or user) input. Getting the control flow right in these cases gets even harder through the Activity semantics concerning `beginExecute()`, because decisions in case distinctions have to be handled correctly (while the Activity is still running), and the handling of errors has to be implemented where the next Activity is called.

These difficulties suggest a model-based description that concentrates on the idea behind reactive systems and abstracts from implementation details such as the execution semantics seen with Activities. In mechatronics and robotics, such behavior is often described through the concept of state machines [10, 36, 58], describing system states and transitions in a graphical or textual way. One way is to specify and implement the state machine directly in application code. Here, applying the state pattern [31] or using state machine execution engines [39, 96] can help, however with limitations and still fighting with the specialties of the Activity semantics.

To simplify the process of state machine definition, **StateChartActivities** can be used (cf. figure 7.9). Structurally, they consist of States representing Activities, and Transitions that are triggered by `RealtimeValues`, `ActivityResult`s or exceptions. For each **State**, an Activity factory has to be defined that creates an Activity instance that is to be executed when the State is entered. The factories are required because an Activity can only be executed once, while a State can be entered multiple times. For **Transitions**, an originating State, a destination State and a Condition are given. The **Condition** consists of a filter on the `ActivityResult`s, implemented as a predicate to decide if a given `ActivityResult` of the Activity created for the originating State triggers the Transition. `ActivityResult`s here represent completion as well as error situations, and situations where the Activity can be taken over, so that all three cases can be used to cause a Transition. Additionally, the condition can contain a guard given as a `RealtimeBoolean` that has to be *true* to take the Transition, e.g. to react to external events. To define a `StateChartActivity`, the affected Devices and a start State have to be given. Then, Transitions to further States can be added, interpreting Transitions with a *null* destination as transitions to an implicit final State. The `StateChartActivity` is able to take over the previous Activity using its start State Activity, and allows to be taken over by a following Activity through State Transition to *null*. However, taking over an Activity (from outside or within the state chart) is only allowed if both Activities control



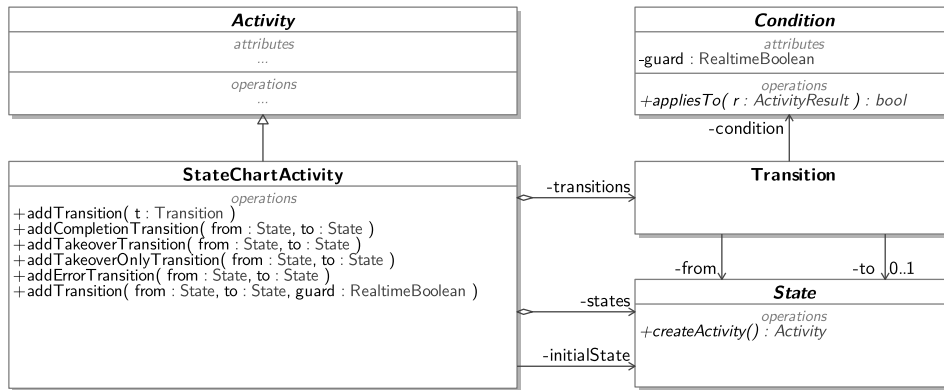


Figure 7.9: Classes used with StateChartActivities

the same devices – otherwise, Transitions are only allowed for ActivityResults that are marked as done.

The implementation of StateChartActivity is similar to the one of ThreadActivity. First, it uses the initial State to create an Activity, which is prepared for the ActivityResults of the predecessors. The possible results provided by the first Activity are filtered by the predicates given in the Transitions to decide for which cases a Transition is to be taken. For all States reachable through a Transition, a corresponding Activity is created, and prepared with the ActivityResults of the first Activity corresponding to the Transition, amended with additional conditions if given in the Transition. The resulting ActivitySchedules are then loaded to activate the Transitions. Once a Transition has been taken, the same process is repeated for the outgoing Transitions of the new State, preparing the Activitys for the following States using the ActivityResults of the new State's Activity. If a Transition with *null* destination is reached, the corresponding results are provided as possible results of the StateChartActivity. The Activitys for the individual States are prepared in the parallel thread to avoid blocking the event handler thread, thus avoiding sensor processing lags.

This way, a state chart can be implemented, without real-time guarantees for the state transitions, but on a best-effort basis similar to ThreadActivitys or pure control flow solutions. Transitions are guaranteed to be taken if their condition happens late enough so that the following state has been prepared, while otherwise the previous State remains in control until a successor is ready.

To increase the probability for Transitions to be taken on short Activitys, the concept of predictive loading can be introduced: Instead of preparing the Activitys for the following State and its Transitions, it is possible to already prepare Transitions that lead to further States. This way, a bounded look-ahead of  $n$  can be defined, so that all States that can be reached through  $n$  Transitions are prepared and loaded. During execution, some steps not taken can be unloaded, while further steps that become reachable in the bounded distance have to be prepared. However, a limit is required as a number of steps to look ahead, as a number of

times each State may be visited during preparation, or based on more intelligent algorithms taking into account the expected execution time of the State Activities or transition probabilities, otherwise cycles in the Transitions lead to infinite prepare times. Another point to note is that while increasing  $n$  reduces the risk of coming to a Transition that has not yet been prepared, the amount of preparation and thus the preparation time grows exponentially based on the number of outgoing Transitions, though many steps prepared are never executed, and time can be missing to prepare the chosen path. Thus, a sensible limit has to be chosen, balancing the amount of unnecessary preparation and the risk of reaching unprepared States.

Concrete examples of StateChartActivities are given in chapter 10, where they are used to implement the case study applications. Synchronizing the work of multiple Devices within one StateChartActivity and reacting to events created in others is there used as means of workflow synchronization.

## 7.4 Comparison with Previous Work on the Robotics API

Compared to the work of Angerer [2] and Angerer et al. [3], the semantics and capabilities of Activities were slightly adapted: While the definition as “a real-time critical operation, affecting one or more Actuators, that supplies meta data about the state of each controlled Actuator during or after the execution of the operation” [2, p. 121] still holds in general, the exact amount of real-time criticality changed: Previously, an Activity was allowed to have a deterministic (e.g. for a planned one) or non-deterministic execution time (e.g. for switching controllers), however the expected outcome and its metadata had to be known in advance. This allowed to pre-plan any composition of Activities, guaranteeing deterministic transitions (although the inner duration could be indeterministic). In contrast, the new approach allows Activities to provide (some of) their outcomes later at run time. This limits transitions in these cases to a best-effort semantics, however provides more flexibility by allowing ThreadActivities and StateChartActivities to encapsulate longer sequences and reactive behavior for mobile robots (cf. design goal 7.5). Meanwhile, it is still possible to detect whether an Activity provides the results in advance and transitions are thus guaranteed for compositions, if absolute determinism is required.

This change was complemented by a shift towards Activities representing Actuator capabilities that provide all real-time guarantees required for the tasks, and that can maintain their result state for an indefinite time, but allow gaps between execution if the application cannot provide successive Activities in time (while the intermediate result state is reliably maintained). For the common composition methods of parallel and sequential composition, the determinism guarantees remain unchanged when used with “classic” Activities that know their results in advance, while additionally allowing to combine the “new” type of Activity in a best-effort fashion (cf. design goal 7.4).

With respect to the metadata contained, the expressiveness has been extended by introducing

multiple results with metadata about their state. This way, common concepts (such as the robot position when a result has been reached) can be reused for different results (e.g. for the state when motion blending should be applied and the completion state), instead of introducing individual metadata properties (*CartesianTargetProperty* and *BlendingCartesianStateProperty* in [2]). Additionally, it becomes possible to specify multiple different outcomes with their metadata, instead of having to classify all but one as error cases (which e.g. had to be done for canceled motions). Using synchronization rules as an underlying mechanism (instead of a single Command that can be taken over if supported by the following Command), it is furthermore possible to directly distinguish between different outcomes, without having to check sensor data or the environment state, simplifying Activity design and allowing to handle different results independently.

When working with a sequence of two Activities modeling a blending motion, the previous design required the second Activity to have the same results regardless of the first Activity being taken over during motion or after completion. This was required because the second Activity was only allowed to provide one *CartesianTargetProperty* and one *BlendingCartesianStateProperty*, which thus had to be valid for the normal and the blending case, strictly limiting how the blending motion and its motion progress (which was typically used as a blending condition) had to be interpreted (e.g. 80% of the normal motion and of the blending motion had to be at the same position and velocity). Through the use of multiple ActivityResults with their metadata, this restriction is no longer in place, because the different ActivityResults may be provided for the normal or blending case. However, supporting outcomes with different metadata exponentially increases the amount of different results that can happen in a sequence, because for each new predecessor result a set of new results has to be created by each Activity. To cope with this situation, the newly introduced ThreadActivity and StateChartActivity work with a limited planning horizon and can thus avoid the combinatorial explosion experienced with longer sequences.

Looking at the Activity implementation, an Activity is no longer mapped to a single *TransactionCommand*, but to a complete Command structure along with the corresponding metadata, represented by an ActivitySchedule. This leads to smaller Commands reducing execution overhead, and additionally allows to incrementally extend the resulting Command structure when new Activities are executed. Furthermore, through using CommandResults and synchronization rules it is now possible to take over two preceding running Activities from one successor, which had not been possible with the previous *ActivityScheduler*, but occurs within mobile manipulators where platform and arm can be controlled independently but also combined.

When implementing service-oriented automation as introduced by Hoffmann [39], the newly introduced ThreadActivity and StateChartActivity can be seen as an appropriate result for *Task* and *Skill* services: Within these composed Activities, it is possible to encapsulate bigger tasks

including case distinctions that are decided in the application context based on the situation context and sensor data, and to provide them as an Activity. This way, *Skills* and *Tasks* provide results that can be processed further, and that fully support features such as motion blending into and out of *Skills*, instead of relying on a coordinator at a higher level to make sure that using `beginExecute()` in the *Skill* implementation is sufficient and acceptable for the Activity transitions.

## 7.5 Related Work

In classical industrial robotics, robot synchronization is typically achieved through special programming language features, or through a programmable logic controller (PLC) linking and coordinating the different robot controllers. Through *RoboTeam* for KUKA robots, *MultiMove* for ABB and *Independent/Coordinated* for Yaskawa Motoman [32], it is possible to coordinate the motions of different industrial manipulators. These allow to synchronize the start time of motions for different robots (thus providing time synchronization), and to define and execute geometrically linked motions of the corresponding robots (with data synchronization). *RoboTeam* requires one controller per robot, while the other alternatives allow to control multiple robots with a single controller. In industrial robot cells, workflow synchronization between the included robots is typically based on a cell-PLC that coordinates the tasks of the different robots, either by triggering the robot programs to be executed, or by signaling start events through digital outputs and receiving status information through inputs (cf. [39]). However, all these methods require a wired connection between the robots and are thus not applicable to the cooperation of mobile robots.

Looking at device capability modeling and composition, different approaches are available in *OROCOS* and *ROS*. As described in section 6.4, in *ROS* interruptible tasks are modeled through *Actions* based on *actionlib* [28]. These *Actions* can be seen as a way to model device capabilities, while the set of possible *Actions* can be extended by introducing new components that work as Action servers. To combine multiple *Actions*, a new *Action* can be defined in a new component that invokes the corresponding *Actions* in parallel or sequentially (cf. design goal 7.4). However, such *Action* compositions provide no real-time guarantees, as they rely on network communication between the different *Action* servers. Furthermore, different *Actions* have no common interface and thus cannot directly be passed around between different components (such as a planning component that provides its result as an *Action* call to be executed by another component).

To define reactive behavior on task-level, *SMACH* state machines introduced by Bohren et al. [10] can be used with *ROS*. *SMACH* allows to define *States* based on *ROS Actions*, *Services* or Python code, and allows to combine these *States* into new composed *States* in a concurrent, sequential or state-machine form. Therefore, each *State* defines different outcomes

that can be used in composite *States* to define reaction or handle error conditions. Furthermore, *SMACH* states (and thus state machines) can be provided as *actionlib* *Actions* to make composed capabilities available to other components. These composition mechanisms are similar to the ones introduced in this chapter, however with the difference that they do not provide timing guarantees and thus cannot be used for low-level systems that have to switch between states at precisely defined moments. Furthermore, *actionlib* provides no explicit semantics or metadata for *Action* post conditions, making further planning during execution harder to achieve. Similarly, *SMACH* state machines run robot actions in a blocking way, without using the execution time for further (motion) planning steps (cf. design goal 7.3).

In *OROCOS*, reactive behavior can be modeled using *rFSM* state charts introduced by Klotzbucher et al. [58]. With *rFSM*, hierarchical state charts without parallelism can be described, aimed at the coordination concern of robot applications. Following the *pure-coordination* pattern [58], state charts are only used to process events created by monitor components and raise events that handled by configurator components, manipulating the set of active components to achieve the goal. Being implemented in *LUA* with specialized memory allocation and garbage collection, *rFSM* state charts can be executed with real-time guarantees, thus allowing real-time coordination.

Additionally, Scioni et al. [93] describe how to achieve *preview coordination* with *rFSM* state charts: In this approach, some state machine transitions are labeled as likely, giving the execution environment hints about execution probabilities. This way, likely successor states can be prepared (performing some of their work) while the previous state is still active, as long as the preparation steps do not conflict with the actions performed in the current state. This allows to reduce execution time, while keeping the action definitions in a single place (instead of moving the preparation step into the previous state), thus improving reusability.

To a reduced extent, similar effects in the proposed approach can be achieved for mainly sequential workflows using `beginExecute()`, where execution proceeds to the following code, and *Activitys* of disjoint devices can be executed in parallel. In an example of grasping an object using a *youBot*, first issuing a motion of a *youBot* platform using `beginExecute()`, followed by a motion of the arm leads to parallel execution similar to the one suggested in the *preview coordination* example. Furthermore, while not directly aimed at execution, the preparation aspect of *StateChartActivitys* is related and can benefit from similar likelihood annotations to decide which transitions in the state chart will likely happen and should be prepared first to reduce the risk of missed transitions.

While *preview coordination* introduces a form of decoupling between workflow and capability execution, the *rFSM* mechanism does not include semantics descriptions for the results of states. This way, it is not possible to analyze the expected results of a *State* and to prepare for the following tasks using metadata about the expected outcome, a powerful and important feature offered by the *Activity* model introduced in this chapter.



## SENSORS AND OBSERVATIONS

Using the concepts introduced in the previous chapters, it is possible to exactly describe desired robot behavior in a perfect world. This is sufficient in a completely controlled environment, e.g. for industrial robots on a shop floor, where the task is defined with exact position information, and no variability exists. In these situations, for robot control it is sufficient to work with proprioceptive sensors measuring the internal state. However, if variability or uncertainty in the environment exists, exteroceptive sensors have to be used that observe the surroundings.

In this context, Requirement 5 about accessing and using sensor data and Requirement 6 about integrating sensor data into the world model become relevant. Apart from aspects already present in industrial robots, these requirements cover important additional topics for mobile robots. For the software design, these requirements can be expanded into further, more specific design goals: In general, sensor data has to be accessible in an application.

**Design Goal 8.1.** *Provide access to sensor data in applications.*

Different from other `RealtimeValues` (as introduced in section 6.1.1), sensor data can arrive at a lower frequency or delayed due to hardware communication or processing time. This is especially common in mobile robotics, where light-weight sensors often offer lower update rates, and where the exact time when data is measures becomes important and has to be taken into consideration because the sensor itself moves, yielding design goal 8.2.

**Design Goal 8.2.** *Support handling of infrequent or delayed sensor data.*

When using different sources of delayed or infrequent sensor data, some tasks still require to work on consistent data, i.e. data that belongs to the same time. The framework should provide

the ability to do this, especially when combining sensor data with data describing the position of the sensor at the measurement time.

**Design Goal 8.3.** *Allow to access and process consistent sensor data.*

Quite often, sensors are used to measure details about geometric features present in the environment. The raw sensor data thus has to be converted into geometric data.

**Design Goal 8.4.** *Derive geometric information from sensor data.*

This geometric sensor data typically describes coherences that are also modeled in the World model. When such correspondences between sensor data and geometric features of the environment are defined, the data can be consistently integrated into the World model.

**Design Goal 8.5.** *Define the relationship between sensor data and geometric features.*

**Design Goal 8.6.** *Integrate sensor data into the world model.*

Separating the latter two aspects is an important prerequisite to supporting distribution, because the same abstract relationship can lead to different sensor data handling in the robot systems, depending on the further data or knowledge available in the systems. The proposed framework allows to access the data of Sensors that measure the robot state (interoceptive) as well as the environment (exteroceptive), as described in section 8.1. Some sensor values directly or indirectly observe aspects modeled in the World model. These measurements can be integrated to resolve the uncertain or unknown aspects present in the World model. Section 8.2 describes how such observations are modeled and used to fill in unknown information. Finally, Section 8.3 compares this approach to previous work on the Robotics API, and section 8.4 gives an overview about related approaches of handling sensor data.

## 8.1 Accessing Sensor Data

**Sensors** are represented as **Devices** that are able to provide data in real-time. Through a **SensorInterface**, a specialized **DeviceInterface** retrieved by the `use()` method, one or multiple **RealtimeValues** can be accessed that represent the current values measured by the sensor (cf. section 6.1.1).

### 8.1.1 Accessing Individual Sensors

One individual **RealtimeValue** from a **Sensor** can be used in three ways: Its value can directly be accessed from an application, it can be used in further calculations on **RealtimeValues**, and it can be used to define or influence **Actions** or **Commands**.

In the first case, the **RealtimeValue** provides a method **getCurrentValue()** that retrieves the current value from the execution environment and returns it to the application. When



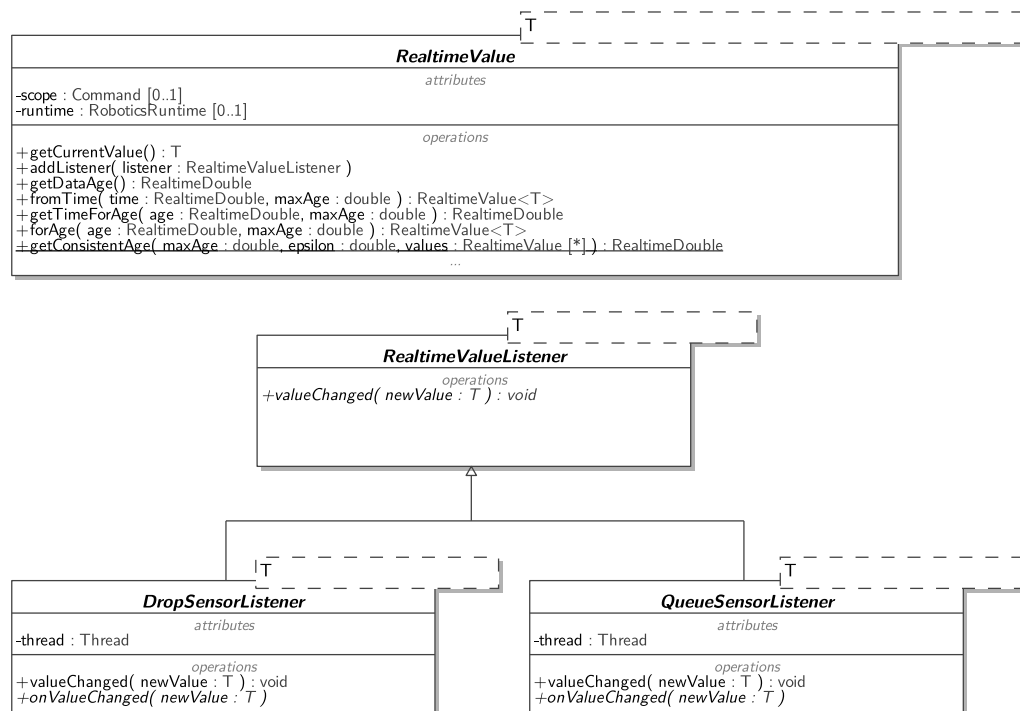


Figure 8.1: Specialized RealtimeValueListeners for handling sensor data

the value is required more often, a **RealtimeValueListener** can be added that will be informed when the sensor value changes. By default, the **RealtimeValueListener** is informed from the main thread communicating with the execution environment. Thus, the **RealtimeValueListener** should not perform lengthy calculations, otherwise all further sensor processing is blocked and a considerable time lag can be introduced. When long operations are required on sensor data, two specialized classes of **RealtimeValueListener** exist for different use cases (cf. figure 8.1), using individual event handling threads: The **DropSensorListener** uses its own thread to process the arriving sensor data. If further sensor values become available while the previous data is still being processed, all data except for the newest is ignored. Thus, the **DropSensorListener** always performs its calculations on the latest data, however may miss some intermediate values. If sensor data should not be lost, the **QueueSensorListener** can be used. It keeps a queue of sensor data that has been received but not yet processed. The data is processed in a thread that sequentially handles all sensor data added to the queue. This way, it processes all received data, however the newly received data can be far ahead of the processed data, so that findings of the processing can already be outdated.

Furthermore, raw Sensor data in the form of **RealtimeValues** can be used to perform calculations as introduced in section 6.1.1, computing derived values that interpret the data. Therefore, the concrete types of **RealtimeValues** provide methods that allow to apply arithmetic operations,

True value	A	B	C	D	E	F	G	H	I	J	K	L	M
Ideal sensor value	A	B	C	D	E	F	G	H	I	J	K	L	M
Data age	0	0	0	0	0	0	0	0	0	0	0	0	0
Infrequent sensor value	A	A	A	D	D	D	G	G	G	J	J	J	M
Data age	0	1	2	0	1	2	0	1	2	0	1	2	0
Delayed sensor value			A	B	C	D	E	F	G	H	I	J	K
Data age			2	2	2	2	2	2	2	2	2	2	2
Infrequent delayed sensor value			A	A	C	C	E	E	G	G	I	I	K
Data age			2	3	2	3	2	3	2	3	2	3	2

Figure 8.2: Timing behavior of different sensors

combining the value with other `RealtimeValues`. The result of such calculations is calculated in real-time when accessed and again a `RealtimeValue` itself that can be used wherever pure sensor values can be applied.

Finally, `RealtimeValues` from Sensors can be used to influence or define Actions. They can be used in goals specified for a robot, as a correction applied to a given trajectory, in Assignments running in parallel with an Action, and also as cancel conditions that gracefully bring an Actuator to a stop when an undesired value is measured (cf. section 6.1.2).

### 8.1.2 Consistent Combination of Sensor Data

`RealtimeValues` from sensors provide additional information about the time when the values were measured. This is available through the `getDataAge()` operation providing a `RealtimeDouble` indicating the age of the sensor value in seconds. This is especially helpful for sensors that do not continuously provide readings, but only sporadically or with a low update frequency, or when preprocessing of the sensor takes a substantial amount of time, so that when a sensor value is reported, it is already outdated by a given amount of time.

Figure 8.2 gives an overview about Sensors with different timing characteristics. The first line gives the real value of the measured variable (ground truth), while the following lines give the values reported by the different sensors and the corresponding data age for the provided data. An ideal sensor always provides the correct value, and thus has a data age of 0. An infrequent sensor however works with a lower update rate and thus returns the same value (with increasing data age) until a new update occurs. Other sensors can need processing or measurement time, and thus provide the data at a later time, yielding a constant data age  $> 0$ . Infrequent and delayed sensors only sporadically provide data, which took some processing time, leading to varying data age. For clarity of presentation, all these sensors in the example share the commonality that they provide exact values (i.e. the measured value coincides with the ground truth) and that they offer a correct data age.

Under these conditions, it is possible to compute temporally coherent results when working with multiple `RealtimeValues`: The method **fromTime()** available on `RealtimeValues` allows to access the value of a `RealtimeValue` as it was at a given time in the past. The time is given in seconds, and limited to a maximum amount. This method returns the data that was known at the given time, but not data that might have become known later but refers to the given time. To find data that refers to a given time, **forAge()** can be used that provides the best data for the given time, no matter when it became available. To provide this, it uses the method **getTimeForAge()** that finds the best parameter to **fromTime()** so that the return value of **fromTime()** belongs to the given time. If multiple sporadic sensor values are to be processed consistently, the static method **getConsistentAge()** exists that accepts multiple `RealtimeValues` and finds a data age so that for all sensors data has been available within a given time interval. This time can be used with **forAge()** to access the corresponding values. This way, it is possible to perform calculations using an infrequent sensor value, together with values of other frequently available `RealtimeValues` from the time the sporadic sensor value was valid.

Working with the sensors from figure 8.2 at the last shown time instant ( $M$ ), **getDataAge()** returns 0 for the ideal and infrequent sensor, and 2 for the delayed sensors. Using **fromTime()** with a time of 3 units provides the values  $J$ ,  $J$ ,  $H$  and  $G$  and is thus not sufficient for a consistent snapshot. With **getTimeForAge()** for an age of 3, the values 3, 1, 1 and 1 are returned, representing the latest time that provided a value  $\leq J$ . Correspondingly, **forAge()** with an age of 3 can return  $J$ ,  $J$ ,  $J$  and  $I$ , taking the value at the time given by **getTimeForAge()**. Finding a consistent time for all four sensors with **getConsistentAge()** and an allowed time interval of 0 returns 6 representing the age of  $G$ , the last value that has been seen by all sensors.

More formally and for continuous time, the behavior of **fromTime()**, **getTimeForAge()**, **forAge()** and **getConsistentAge()** can be described mathematically. For this description, the sensor  $s$  measures a quantity that in reality has the value  $trueValue(s, t)$  at time  $t$ . Discretizing time into intervals of  $\Delta t$ , for time step  $i$  that corresponds to the time  $t = \Delta t \cdot i$ , the sensor reports the value  $value(s, i)$ . To describe the relationship between measured  $value()$  and  $trueValue()$ , the concept  $dataAge(s, i)$  is introduced that describes age of the value returned for time step  $i$ :

$$value(s, i) := trueValue(s, \Delta t \cdot i - dataAge(s, i))$$

The time for which data is reported by the sensor is assumed to be monotonous, so the following restriction has to hold for each sensor  $s$  and time instant  $i$ .

$$dataAge(s, i + 1) \leq dataAge(s, i) + \Delta t$$

The operations  $fromTime(s, i, a)$ ,  $timeForAge(s, i, a)$  and  $forAge(s, i, a)$  accept a time span  $a$  in seconds, speaking about the past moment  $\Delta t \cdot i - a$ :

$$fromTime(s, i, a) := value\left(s, i - \left\lceil \frac{a}{\Delta t} \right\rceil\right)$$

$$\begin{aligned} \text{timeForAge}(s, i, a) &:= \Delta t \cdot (i - \max\{j \in \mathbb{N} \mid j < i \wedge \\ &\quad \Delta t \cdot j - \text{dataAge}(s, j) \leq \Delta t \cdot i - a\}) \end{aligned}$$

$$\text{forAge}(s, i, a) := \text{fromTime}(s, i, \text{timeForAge}(s, i, a))$$

To calculate a consistent time for a set  $S$  of sensors,  $\text{consistentAge}(S, i, e)$  can be used:

$$\begin{aligned} \text{consistentAge}(S, i, e) &:= \Delta t \cdot i - \max\{t \in \mathbb{R} \mid t < \Delta t \cdot i \wedge \\ &\quad \forall s \in S \exists j \leq i : t - e < \Delta t \cdot j - \text{dataAge}(s, j) \leq t\} \end{aligned}$$

To process consistent values of multiple sensors in an application, it is possible to define batches of `RealtimeValues` as **RealtimeTuples**. Using these `RealtimeTuples`, a listener can be added that receives tuples of values that became available at the same time. In mobile robotics, a typical use case of `RealtimeTuples` is to combine sensor data with position information about the sensor (in conjunction with `getConsistentAge()` and `forAge()` for delayed sensors). This way, consistent pairs of sensor position and sensor readings can be processed, allowing to interpret the sensor data in the correct geometric context. Figure 8.3 gives two examples of sensor value combinations: In *nowAndThen*, the value of *value* is provided for two different times (now and 2 seconds ago), while *consistentPair* provides the a pair of *position* and *value* for which the measurement time differs by less than 0.1 seconds (if such a pair has been measured in the last 2 seconds).

### 8.1.3 Sensors in the Case Study

Looking at the Actuators used in the case study, different sensors can be found: The youBot platform provides `RealtimeDoubles` for the angular position and velocity of each wheel. These can be transformed into a `RealtimeTransformation` and `RealtimeTwist` describing the displacement of the wheel rigid bodies relative to the platform *Base*. Additionally, the youBot platform provides sensor data concerning the position of its *base* relative to its *odometry origin* as a `Pose` and `Velocity`. These values are available as the *measured* position, calculated based on the wheel revolutions measured through the integrated encoders and using the kinematics function describing the correlation to Cartesian space. If the position of the youBot platform is controllable, another instance of this information is available: The *commanded* position and velocity, which has been given as a set-point for the current time instant by the application. The controller implemented within or for the Actuator is responsible for making sure that the measured position converges to the commanded position, ideally showing good tracking performance. The youBot arm provides `RealtimeDoubles` concerning the position and velocity of its Joints. Using these values, `RealtimePoses` and `RealtimeVelocitys` are calculated, converting the raw numeric value into a `Transformation` with a rotation around the Z axis by the given amount. For youBots that are controlled in the current system (cf. section 2.3.2), a *commanded* position

```

1 RealtimePose position = ...
2 RealtimeDouble value = ...
3 // access current and historic data ...
4 RealtimeTuple nowAndThen = new RealtimeTuple(value,
    value.fromTime(2));
5 // create a consistent pair ...
6 double maxAge = 2;
7 double epsilon = 0.1;
8 RealtimeDouble time = RealtimeValue.getConsistentAge(maxAge, epsilon,
    value, position);
9 RealtimeTuple consistentPair = new RealtimeTuple( value.forAge(time,
    maxAge), position.forAge(time, maxAge));

```

Figure 8.3: Java code combining sensor values

and velocity is available that describes the latest position and velocity set-point of the youBot, while for youBots from other systems only measurements or observations may be present. The youBot gripper has no integrated sensors, and thus only provides a commanded position as a `RealtimeDouble` that is converted into Transformations of the fingers.

The quadcopter has a lot of integrated sensors, such as an accelerometer, gyroscope and magnetometer. However, in the example no telemetry channel is used to transfer data from the quadcopter to the controlling computer. Thus, these values are not available to applications.

Apart from sensing functions of Actuators, dedicated sensors appear in the case study: The *Hokuyo* laser scanner measures the distances to obstacles within an angle of  $240^\circ$ . Each measurement takes 100 ms, so sensor values are provided at a rate of 10 Hz. These readings are provided as a `RealtimeDoubleArray`, holding 683 distances expressed in meters to be interpreted as polar coordinates. Additionally, they can also be provided as `RealtimePoints`, describing the obstacles seen for each angle as a `Point`. These points are described based on the *ScanFrame* of the laser scanner, while using the array index as well as *minAngle*, *maxAngle* and *pointCount* to compute the Cartesian positions belonging to the measured distance.

The *Vicon* tracking system allows to find the Cartesian position of configured objects equipped with markers, as well as of markers that do not belong to an object. For configured objects (called *ViconSubjects*), it offers a `RealtimePose` and `RealtimeVelocity` describing the position and motion of the object, expressed in its *Origin* Frame. Within the *Vicon* system, the detected markers are checked against the marker configurations of the known object. Once a corresponding marker pattern is recognized, the *Pose* is updated. If an object is not detected, e.g. if the object is currently not in view, the old position is held, and the data age is adapted appropriately. For markers that do not belong to an object, `RealtimePoints` are provided that give the position of the markers expressed in the *Origin* Frame. Using only a single marker, it is impossible to calculate an orientation, so no full pose can be provided. Additionally, individual markers have no identity and thus cannot be recognized as a specific instance. To still handle

them in a meaningful way, the unlabeled marker tracking remembers the last known position of the marker, and chooses the unlabeled marker closest to the previous position to update the `RealtimePoint`. As a default, tracking is performed at 100 Hz with a lag of about 5 ms, while the frequency can be increased to over 500 Hz.

These sensor values can be manually processed in an application, however many of them describe aspects in the World model and can thus be integrated there. This is where Observations come into play.

## 8.2 Integrating Sensor Data into the World Model

The physical environment of a mobile robot consists of various objects, each at its respective position. Depending on the amount of structure in the environment, some of these positions are constant and can thus directly be modeled as geometry of a robot application. Other positions however change over time and thus cannot be modeled exactly for an application that should be reusable later. Still, in order to work with them the application has to know about the existence of these objects, as well as about their logical relationships to further objects (“I expect a work piece to be in this room.”). While the objects do have an exact position in the physical world, the application initially has no precise position information, limiting the amount of interaction that can be performed with these items.

To improve this situation, sensing can be employed to give the robot a glance of its environment. Based on sensor data, some positions (and velocities) of objects can be recovered, contributing to the geometry knowledge of the application. To facilitate this, the logical relationship between sensor measurements and geometric aspects is required, describing what is measured by the sensor. At application run time, these relationships and incoming sensor data can be used to update the unknown or uncertain parts of the World model, so that it reflects a consistent interpretation of the received sensor data.

In software, this process is performed through the introduction of uncertain Relations, Observations and Estimators. In application geometry as well as in Device definition, some Relations have an unknown or uncertain Transformation. Some of the used sensors directly measure one of these Transformations, while others talk about geometric aspects that affect multiple Relations. To describe this aspect, the World model is extended by the definition of Observations (cf. section 8.2.1). Based on these Observations, an Estimator can add its estimation about the unknown Transformations to the World model, deriving ways how to process sensor data in order to recover geometric information about the unknown positions (cf. section 8.2.2).

### 8.2.1 Observations – Describing what is Measured

An **Observation** describes that a certain aspect of the World model is measured by a given sensor, or more generally that it is available as a given `RealtimeValue`. A typical kind of Observation is

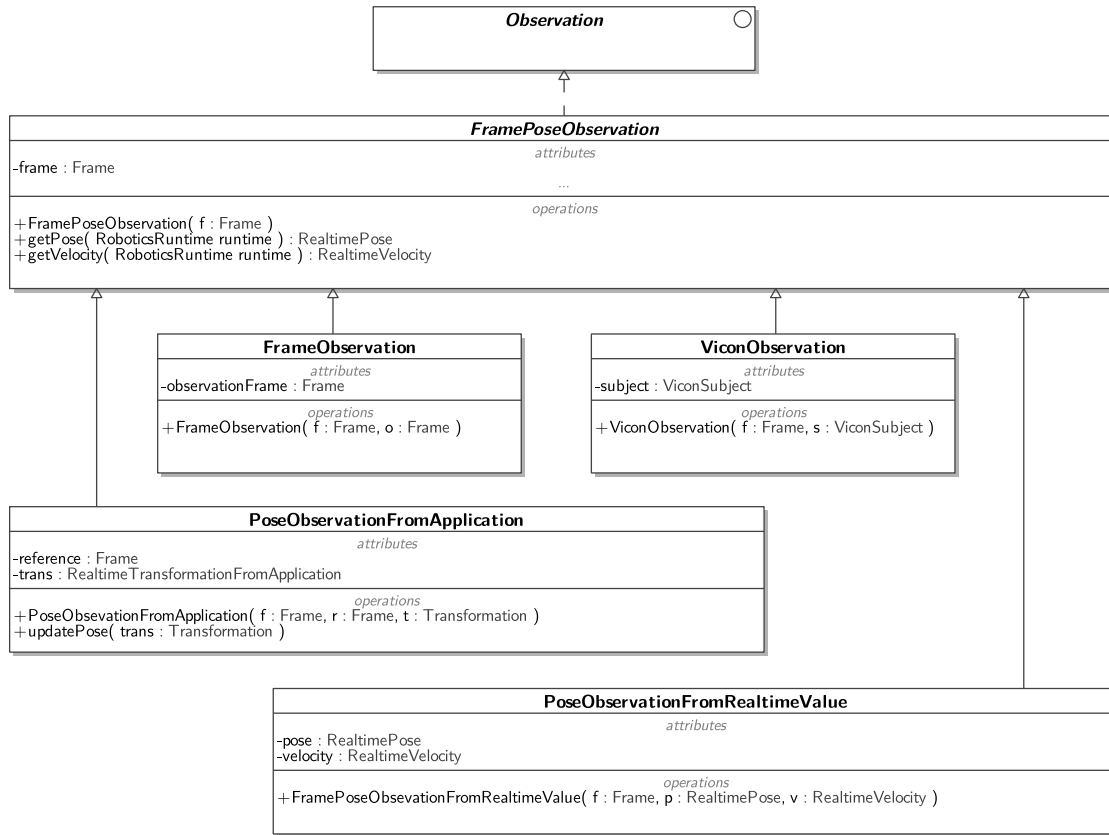


Figure 8.4: Observations used to integrate sensor data into the World model

the **FramePoseObservation** defining that a Frame's place and motion can be expressed through a given `RealtimePose` and `RealtimeVelocity`. Sensor based `RealtimeValues` are often limited to a specific execution environment (cf. section 9.1.2), so the getters for the Pose and Velocity are provided with the `RoboticsRuntime` to provide the value for. A **FrameObservation** (cf. figure 8.4) describes that the position and orientation of a Frame is given by another Frame called the *observationFrame*, which may be provided by a sensor system. The **FrameObservation** can be expressed as a **FramePoseObservation** that gives an identity Transformation and Twist relative to its *observationFrame*. To simplify work with the Vicon tracking system, a **ViconObservation** is provided that is based on a **FramePoseObservation**, and specifies that the position of a Frame in the World model is tracked by the given *ViconSubject*. Additionally, the **PoseObservationFromApplication** allows to handle cases where more complex sensor processing is required to find the Pose of a tracked object, which is performed in the application logic. Using this kind of Observation, the application can provide Pose information for a Frame whenever its computation completes, which can then be used in the World model. For sensors that provide their data as `RealtimeValues` and further `RealtimeValue` sources, the **PoseObservationFromRealtimeValue** allows to specify an Observation that describes position and velocity of an observed Frame as

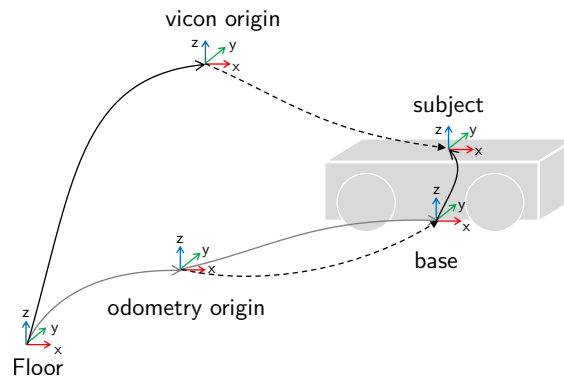


Figure 8.5: Excerpt of the World model for a youBot platform tracked using Vicon Tracking. Solid black lines denote known connections, gray lines label unknown Relations and dashed lines represent Observations.

individual *RealtimeValues*. If the *RealtimeValues* are limited to one certain execution environment, this type of Observation is only valid in this execution environment as well, and does not provide helpful information for others.

When looking at the unknown Relations inside Actuators, most of them can be measured through proprioceptive sensors. These sensors can be modeled using Observations that follow the geometric structure of the object: In the case of the youBot base, four *FramePoseObservations* can be defined that describe the Pose of the wheel Frames relative to the wheel mount Frames. They use a *RealtimePose* which converts the wheel position available as a *RealtimeDouble* into a rotation around the Z axis. Similarly, the connection from the youBot *odometry origin* to its *base* is provided through the odometry sensor and can also be handled as a *FramePoseObservation*. For these Observations that structurally follow the World model by defining a Transformation modeled in a Connection, it is possible to directly create a new Relation that is variable, persistent and known, and takes its Transformation and Velocity from the Observation or its corresponding sensor.

For the second type of unknown Relation however, this is not the case: Assuming the youBot platform is driving on the floor starting at a position not exactly known, application geometry is usually modeled as an unknown Relation from the *Floor* Frame to the youBot *odometry origin*. This Transformation however cannot be measured directly, because the *odometry origin* is an intermediate concept that does not have a direct representation in the physical world (at least after the youBot platform has moved). Figure 8.5 gives an overview over this situation. It shows the World model of a youBot platform on the floor that is tracked through a Vicon System. The figure also includes the *vicon origin* Frame as a reference Frame for the Vicon system, and the two Observations available in the system (represented as dashed arrows). The first Observation describes the Pose of the youBot *base* relative to its *odometry origin*, while the second Observation talks about the Pose of the youBot *base* relative to the *vicon origin* Frame. The Pose provided by



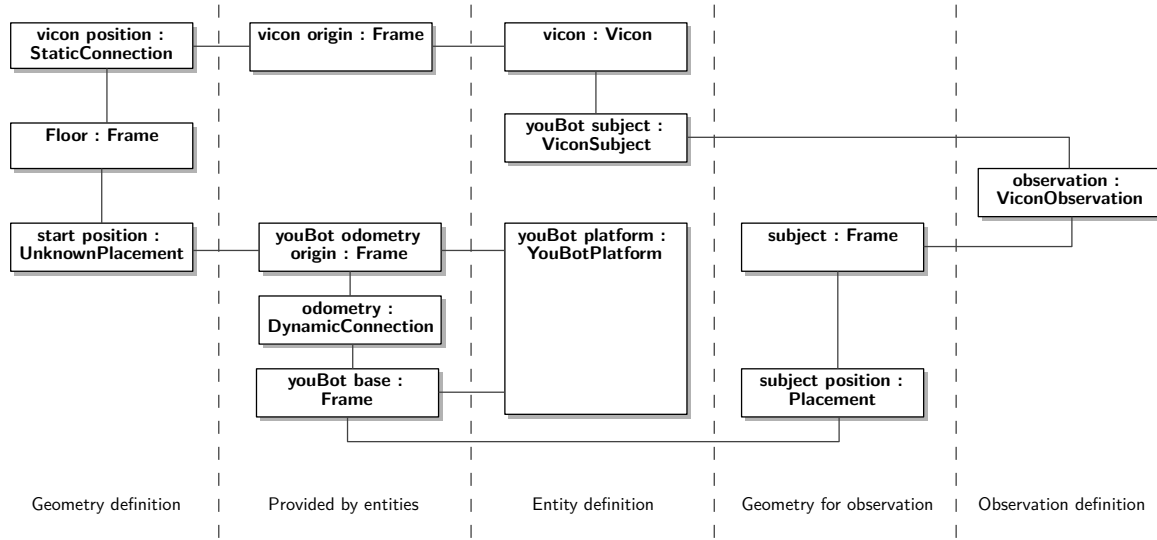


Figure 8.6: Geometry definition for a youBot with Vicon position tracking

the second Observation cannot directly be used to describe the Transformation of a Relation, but first has to be converted.

To define this aspect for an application, an object structure as shown in figure 8.6 is used. In contrast to a fully known case (cf. section 4.3), the geometry definition now uses an *UnknownPlacement* to give the start position of the youBot, and the device configuration is extended by a *Vicon* instance and its *ViconSubject*. Additionally, the *Frame subject* is defined representing relative placement of the *ViconSubject* on the youBot, and a *ViconObservation* defines the correspondence between the *ViconSubject* and the *Frame subject*.

When working with work pieces that are externally tracked, another complicating aspect appears: In a perfect simulation, the work piece lying on the floor is connected to the ground using a known *Placement*. Once a gripper takes the work piece, the *Placement* to the ground is removed and a new *Placement* to the gripper is established. When switching to a more realistic case, the work piece is first connected to the ground through an *UnknownPlacement*. Additionally, an *Observation* is given that provides the position of the work piece relative to the tracking system origin, and thus allows to calculate a *Transformation* for the *UnknownPlacement*. However, once the work piece is grasped, the *UnknownPlacement* is removed and another *UnknownPlacement* to the gripper is added. In this case, the same *Observation* now has to be used to provide another *UnknownPlacement*'s *Transformation*. These cases are handled by *Estimators*, as described in the following section.

### 8.2.2 Estimators – Using Sensor Data to Update the World Model

To coordinate the interplay between unknown Relations and Observations and to provide estimations for the uncertainties present, the concept of **Estimators** is introduced. An Estimator

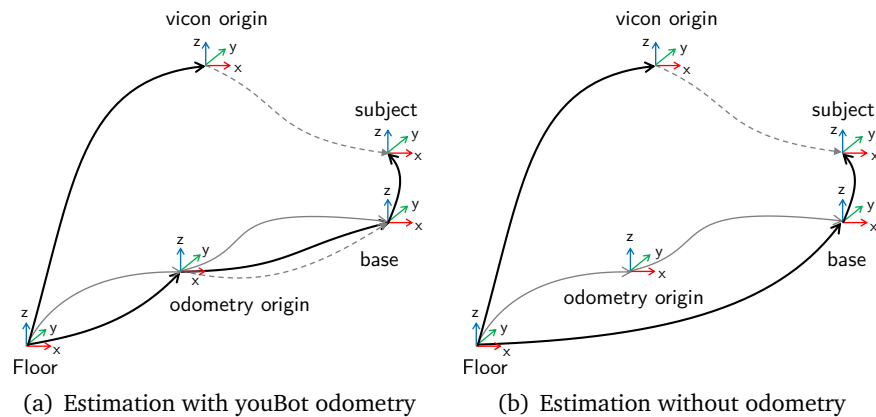


Figure 8.7: Effects of the Estimator for a youBot platform tracked using Vicon tracking

listens to World model changes as well as Observations added or removed, and augments the World model with Relations that run in parallel with one unknown Relation or shortcut multiple ones. For these Relations, a Transformation and Twist is calculated based on the data provided by the Observations.

The easiest variant of an Estimator called **SimpleEstimator** assumes that all Observations are precise and provide values all the time, without any time delay. Given the example in figure 8.5, the Estimator creates two Relations (cf. figure 8.7(a)): The first Relation goes from the *odometry origin* to the youBot *base* and takes the value from the odometry sensor as a Transformation. The second Relation from *Floor* to *odometry origin* is a bit more complex: The Transformation can be combined from the Transformation from *Floor* to *vicon origin*, followed by the Transformation provided by the Observation, the Transformation from the youBot *subject* to the *base* and the Transformation from *base* to *odometry origin*. For the last part, the Transformation provided by the first estimated Relation has to be used. If however the youBot base belongs to another system and is thus not controlled, but only observed from outside, the odometry Observation is not available. In this case, the Estimator has to create a Relation directly from the *Floor* Frame to *youBot base* (cf. figure 8.7(b)).

However, if Observations are only provided sporadically or delayed, an extension to this Estimator is required, called the **TimeAwareEstimator**. It still assumes that all Observations are precise, however accepts that some Observations are only provided infrequently, but with correct time stamps (as seen by the Vicon system when tracking of an object is temporarily lost or a lag in wireless network communication occurs). It further assumes that all unknown Relations are either constant or keep a constant velocity. For constant Relations such as *UnknownPlacements*, it thus assumes that they only have to be changed to correct measurement errors (which is true for objects placed on the ground, as well as for the *odometry origin* of a youBot base), while for variable Relations such as *DynamicConnections* extrapolating with constant velocity is an appropriate estimation. In this case, the Relations are created and the Transformation is

calculated in the same way as by the `SimpleEstimator`, however using `Transformations` from a consistent time (cf. section 8.1.2) and extrapolating it if required. This Estimator can also handle a `PoseObservationFromApplication`, which can for example be used to track an object that is placed on the ground using a laser scanner.

While supporting more use cases, the `TimeAwareEstimator` has the disadvantage of increased memory usage and computation time: To calculate estimations for a consistent time, it has to keep track of previous values and times of the sensor data, and has to select a corresponding time to use the data from. Especially on resource-constrained systems, using a `SimpleEstimator` instead can thus be a better option when no greater time delays are to be expected for the sensor data.

At application run time, Estimator implementations work as a listener that reacts to changes in the World model or Observations. The Estimator tries to find cycles in the graph formed by (known and unknown) Relations and Observations, and uses their information to build new known Relations (called estimations). In cycle search, it tries to minimize the number of unknown Relations required to form estimations in order to keep the estimations structurally as close to the Frame graph as possible. The resulting cycle then consists of an Observation, a (maybe empty) sequence of known Relations, followed by a sequence of known and unknown Relations (that will be estimated and should thus be as short as possible), and a (possibly empty) sequence of known Relations closing the loop. For a found cycle, the Estimator takes the Observation's Transformation and Twist and converts it for the Frames forming the start and end of the unknown Relation sequence, so that `RealtimeValues` describing the overall behavior (position and velocity) of the unknown Relation sequence are available. These `RealtimeValues` are then used to define the estimation Relation, and subsequently evaluated whenever this aspect of the World model is accessed by a motion or directly from the application.

When an unknown Relation is added, an Estimator checks if any of its known Observations can be used to form a cycle in the World model including the new Relation. If so, an estimation Relation can be established based on the Relation and Observation. For known Relations added (that were not created as estimations), the Estimator checks if the new Relation has an effect on any of the existing estimations. This is the case when the Relation influences the first or last unknown Relation resolved through the estimation, causing the estimation to be recreated based on the new situation. When an unknown Relation is removed, the corresponding estimations also become invalid and are removed. Similarly, removing known Relations can invalidate estimations if they occurred in the cycle used to build the estimation.

Adding an Observation may allow to resolve one of the unknown Relations by forming a cycle including the new Observation and building a corresponding estimation. If the resulting estimation contains a subset of the unknown Relations used in another estimation, the latter estimation is removed and recreated with the option to use the newly built estimation, bringing the estimations closer to the Frame graph. When an Observation is removed that has been used by

estimations, the corresponding estimations are removed and new cycles for the corresponding unknown Relations are searched.

Further types of extensions to the Observations and World model are possible that can be handled by more specific Estimators:

- The first extension is towards Observations that do not give the full Pose of a Frame: One sensors might measure the distance to a given landmark, yielding an Observation that describes the distance between two Frames. Another sensor might be able to determine the height of a quadcopter, e.g. through ultrasound or barometric pressure. Here an Observation could only provide the Z coordinate of a Pose.
- Many sensors exhibit sensor noise that is too big to be neglected. It then has to be handled, e.g. by modeling the Observation as distorted by Gaussian noise with a given covariance.
- Some dynamic connections specify constraints about the possible motion, e.g. as a mapping from configuration space to Cartesian space. For example, a robot joint can describe that it maps its single variable to a rotation around the Z axis, while the other rotations and translations are constant. Using these constraints, the number of variables in the system model that have to be estimated can be reduced.
- A description of the dynamics of a dynamic Relation can be used to update the system model at times when no Observations are available. Apart from the static assumption (that the Transformation will not change for constant Relations or will exhibit constant velocity) used in the existing solutions, other assumptions (that are more specific to the system modeled in the Relation) could be used to better describe the expected behavior of the Relation.

In all these cases, standard estimation techniques such as variants of the Kalman filter (as described in section 8.4) could be used. There, the Relations have to be modeled in the system model, while the Observations form the measurement model, yielding the prediction and update steps required to perform the estimation, and providing the estimated Transformations of the unknown Relations present in the World model.

In summary, the introduced concepts of Observations defined on the World model and Estimators to integrate the sensor data as geometric information can be seen as a powerful modeling tool for object-oriented robot programming, allowing the specification of relationships that can be used by different estimation techniques. Apart from continuous computations at run-time, the modeled relationships can additionally be used for offline-processing, allowing to perform parameter estimation based on recorded sensor values through non-linear optimization. Possible use cases here are to determine the exact Pose of a *ViconSubject* relative to the object it is attached to, or the calibration error of a robot, by performing certain motions, recording the

sensor data and optimizing the system parameters in a way to minimize the difference between observations and system model.

### 8.3 Comparison with Previous Work on the Robotics API

The approach proposed in this chapter significantly extends the sensor processing capabilities available in the Robotics API. In the work of Angerer [2], it was already possible to use sensor data to modify trajectories or to define trigger conditions, and to access current sensor values from an application (cf. design goal 8.1). These ways of accessing and using sensor data are similar to the new approach, however the previous approach expected that all sensor data was available without time delays, and at a high update rate, because no precautions were taken to ensure the temporal consistency of the sensor values. As an important extension for mobile robots and handling infrequent sensor data, the new approach allows to access consistent data from multiple sensors, while taking into account the time for which the sensor data is provided (cf. design goal 8.3). Therefore, the sensor data age is introduced, along with ways of accessing sensor data from a given time or for a given age. This is required in many cases, especially when multiple independent values are required to correctly interpret a sensor. In the case of moving sensors, it is vital to know the sensor position along with the sensor measurements to correctly express the position of a sensed object, both of which can be available independently and at different update rates.

In the application example *Factory 2020* [2], sensor data was used to determine the position of the work piece carrier using torque sensors integrated into the used robot arm through approaching from different sides and detecting contact. Furthermore, the joint position sensors of the robot arms were used to model the geometric robot structure by introducing Dynamic-Connections between the robot links, and within the proprietary robot controller for the KUKA Lightweight Robot sensor feedback is used to achieve impedance control of the arm. With this kind of processing, the sensor data directly corresponds with the unknown aspects to be measured. As an extension, the new approach introduces unknown Relations, as well as Observations and Estimators as separate concepts to model the relationship between sensor data and geometric features (cf. design goal 8.5). Separating the measurement descriptions (Observations) from measured variables (Relations) and sensor data processing (Estimators) allows to make the different aspects explicit and model them independently, improving reuse w.r.t. different sensors and leading to more consistent task descriptions. Additionally, it allows to directly react to topology changes in the world model, e.g. correcting the processing of external position tracking data when an object is grasped. This way, applications that rely on geometric sensing can be implemented in a way similar to previous applications that approached the same task in a world with exactly known positions, moving the sensing and estimation aspects into the geometry and deployment configuration.

## 8.4 Related Work

Accessing and processing sensor data is a common task in robotics frameworks and control theory. In both *OROCOS* and *ROS*, sensor data is made available through sensor components that publish current measurements using *Ports* or *Topics*. This data can then be processed and combined by further components that are linked to the corresponding data sources, and used within the component or provided to further components. While the general scheme of processing data is similar, the existing components for *ROS* and *OROCOS* typically have a coarser granularity than the sensor data combination operations in the proposed approach. This is required because the computation overhead introduced through components (with individual threads) and communication, as well as the propagation delay of data transferred through several components limit the amount of components effectively usable in a given scenario.

When combining data from different sensors, *ROS* offers the *message\_filters* library [26] to pre-process and filter messages. It allow to process consistent tuples of data based on the timestamp provided by the corresponding *ROS* messages. The *TimeSynchronizer* merges up to 9 incoming messages from different sources and provides them to a single callback with the corresponding number of parameters. Similarly, the policy-based *Synchronizer* allows to combine messages with exactly corresponding timestamp (through the *ExactTime* policy), or with similar time stamps chosen by an adaptive algorithm (through the *ApproximateTime* policy). When working with geometric data, the *tf* service can additionally provide historical data, and convert transformations using data from the corresponding time as described in section 4.5.

Looking at control theory, the process of integrating sensor data into the world model is similar to the concept of state observers used along with the state-space representation of a system model: A system model consists of a state equation describing the behavior of the system under the influence of the given system inputs, and an output equation that defines the relationship between system state and the measured variables. Standard estimation methods such as the *Kalman Filter* [53] for linear problems allow to process system inputs and measurements to recover the state of the system, tracking the uncertainty of the present state variables. For non-linear problems (that occur in robotics, e.g. through rotations that have a non-linear effect on the robot position when the robot drives forward), extensions of the *Kalman Filter* [51] are available such as the *Extended Kalman Filter* linearizing the problem around the given state, or the *Unscented Kalman Filter* that samples chosen points in the probability distribution. Furthermore, *Particle Filters* [4] allow to tackle problems with greater uncertainty, such as an initial localization of a robot in a known map. These methods can also form the basis of further Estimators for use in the proposed approach, introducing the handling of uncertainty and Gaussian noise.

Looking at *ROS*, the estimation and sensor integration for robot positions is performed through specialized components. According to ROS Enhancement Proposal 105 [71], mobile platforms should be modeled with three distinguished frames: The frame *map* represents the

origin of the robot environment (and a corresponding map), while *odom* represents the starting point of the robot and *base* denotes the robot itself. The transformation between *map* and *base* does not have to be continuous, but may not drift over time, while the transformation between *odom* and *base* has to be continuous, but may exhibit unbounded drift. In the frame tree used in *ROS*, *odom* is the parent frame of *base*, while *map* is the parent of *odom*. This modeling is similar to the model used in the proposed approach, with *odom* representing the *odometry origin* and *map* representing a fixed frame (such as the *Vicon origin* or *World origin*). To handle sensor data, *robot\_pose\_ekf* [72] and later *robot\_localization* as described by Moore et al. [73] provide different estimation algorithms as *ROS Nodes*, including an *Extended* and *Unscented Kalman Filter*. These *Nodes* can process various sensor data, including global positioning systems (GPS), inertial measurement units (IMU) and odometry (ODOM) data, and publish the transformation between *map* and *odom* or the transformation between *odom* and *base*. However, these estimation nodes have to be configured manually, and cannot be used with frame topology changes: When first tracking an object using multiple sensors, and then grasping it using a robot, the parent frame for the object changes, and thus the estimation node has to be reconfigured.

In *OROCOS*, sensor data and estimation can be handled through the *iTaSC* framework [23]. There, uncertainty can be defined for object or feature coordinates, which is then processed during task execution using standard estimation techniques (as introduced above). However, this world model and observation uncertainty model is tied to one given task and is only in effect during task execution. In contrast, the proposed framework allows to globally define the world model, uncertainties and observations independently, allowing to process sensor data also between the execution of different tasks and to derive the uncertainty and measurement model for a given task from the global model.





## EXECUTION ENVIRONMENTS AND DEPLOYMENT

By defining the application geometry, the physical objects and devices used for a robot task are given. However, to actually execute the robot task, it is required to control the physical actuators and read sensor data. Therefore, the actuators and sensors have to be interfaced by a computer system (called execution environment, cf. chapter 6) that communicates with the devices, defines their capabilities and makes them available.

For a set of multiple cooperating actuators and sensors, a single or multiple execution environments can be used, based on aspects such as device connectivity, capabilities and performance. The individual devices have to be assigned to execution environments, defining the deployment structure. When multiple execution environments run on separate computers, the additional aspect of software distribution comes into play, posing further challenges. This is especially important for teams of mobile robots with their inherent distribution aspect.

In Requirement 8, support for multiple devices connected to different execution environments is requested. To achieve this, the execution environments have to be modeled in software and linked to the corresponding devices.

**Design Goal 9.1.** *Model execution environments and their relationship to devices.*

Based on the used execution environment, devices can have different capabilities. Devices that are not assigned to an execution environment cannot be controlled or accessed in an application, while different execution environment implementations can support different features of the device. These features have to be provided to application developers.

**Design Goal 9.2.** *Offer device capabilities provided by the execution environment.*

Some devices are only relevant on a single execution environment, while others are helpful in multiple. For example, shared sensor systems such as the Vicon tracking system can be used by

multiple mobile robots, and for cooperating mobile robots it is helpful to know (some details about) the co-workers through observation or communication.

**Design Goal 9.3.** *Represent the same device in multiple execution environments.*

As the decision which device is attached to which execution environment is primarily a deployment issue, it should not be hard-wired in the application (cf. [64]). Instead, the framework should allow to use a given applications with different execution environment configurations, based on deployment decisions (as long as their used feature set corresponds with the chosen execution environment structure).

**Design Goal 9.4.** *Define deployment independent from application workflow.*

The following sections will go into detail about how deployment and distribution are modeled through configuration (section 9.1), and which different types of execution environments are available for the proposed approach (section 9.2). Section 9.3 compares the approach to previous work on the Robotics API, while section 9.4 describes how other robot frameworks handle these issues.

## 9.1 Deployment and Distribution for Mobile Robots

For deployment and distribution, different levels exist that can be handled individually as described in section 2.3. In the proposed framework, one real-time context is used for each computer that controls devices, and no distribution over multiple computers is used. On the system level, each real-time context is handled as its own system, moving the communication between different real-time contexts to the application level. On the application level, different structuring and distribution schemes can be used, based on functional and non-functional requirements.

The structuring aspects of a robot application on the real-time level are defined as deployment configuration. There, physical devices are linked to real-time capable execution environments, which can then be accessed from applications, as described in the following sections.

### 9.1.1 Connecting Devices to Execution Environments

The execution environment is modeled as a `RoboticsRuntime` that serves as a proxy [31] allowing to access Sensor data and issue Commands. The `RoboticsRuntime` is configured with the data required to contact the execution environment, and contains the implementation to establish that connection. Additionally, it provides status information whether the execution environment is connected and operational, allows to read the current value of Sensors and to be notified

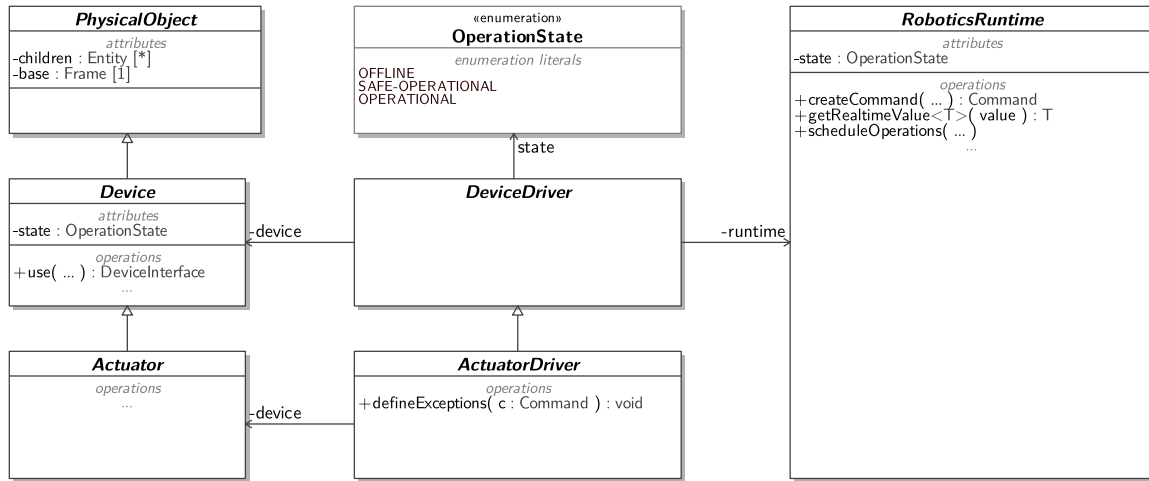


Figure 9.1: Devices and RoboticsRuntimes

about changes in Sensor data, and accepts Commands to be executed by its connected Actuators along with rules when to execute the Commands (cf. section 6.1.3).

To use the capabilities of software Devices in an application, they have to be linked to their hardware counterparts (cf. figure 9.1). This link is provided through **DeviceDrivers** (or **ActuatorDrivers** for Actuators) that connect a software Device to the **RoboticsRuntime** representing its corresponding execution environment. A **DeviceDriver** is specific to the type of Device it controls and to the type of **RoboticsRuntime** it works for. It references the Device as well as the **RoboticsRuntime** and is configured with further data required to identify the hardware device in the execution environment. Through the **RoboticsRuntime**, it provides availability information for the hardware device. Additionally, it encapsulates the knowledge how to retrieve sensor data from and send control set-points to the hardware device within the execution environment. For execution environments following the proposed approach, this means telling which Primitives to use and how to access (and maybe configure) the execution environment. In contrast, the **DeviceDrivers** do not contain logic to directly retrieve sensor data from the hardware device or perform control. They can thus be seen as proxy or adapter objects (cf. [31]).

A Device in an application that does not have an associated **DeviceDriver** just acts as a **PhysicalObject** without further functionality. When a **DeviceDriver** is added that links the Device to its **RoboticsRuntime**, new capabilities or sensor data become available, which are provided through **DeviceInterfaces**. These **DeviceInterfaces** can be **ActuatorInterfaces** for capabilities (cf. section 7.2) and **SensorInterfaces** for sensor data (cf. section 8.1) and are assigned to the Device through composition. Therefore, the Device manages a list of **DeviceInterfaceFactorys** that are responsible for creating instances of the **DeviceInterfaces** once requested by an application through the `use()` method.

When a **DeviceDriver** is added to a Device, an initial set of **DeviceInterfaceFactorys** for the

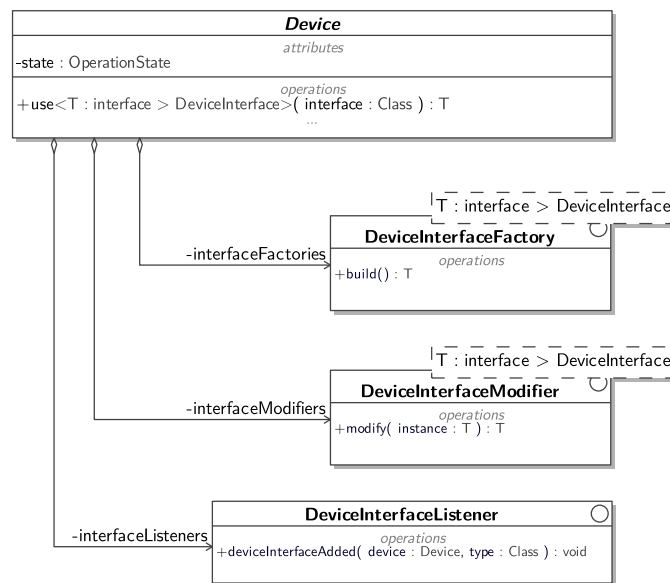


Figure 9.2: Adding and modifying DeviceInterfaces

new features is added to its Device, making available new DeviceInterfaces. Additionally, further dependent DeviceInterface can become available, e.g. for complex tasks that are composed of the capability just added, or for sensor data converted into another format. To allow the definition of dependent DeviceInterfaces, the Device manages DeviceInterfaceModifiers and **DeviceInterfaceListeners** in addition to DeviceInterfaceFactories (cf. figure 9.2).

To access a Device capability within an application, an instance of the corresponding DeviceInterface has to be retrieved through the `use()` method. The DeviceInterface implementation manages a set of configuration parameters, and provides methods for the individual capabilities which then respect the configured parameters, together with parameters stored in the Device and further parameters given with the method call. To provide the DeviceInterface, the Device uses a **DeviceInterfaceFactory** to create a new instance of the requested DeviceInterface. This is required because DeviceInterfaces are stateful (e.g. containing the default DeviceParameters to be used), and thus separate instances of the DeviceInterface have to be provided to independent parts of an application.

To add a new capability to a Device, it is possible to either introduce a new DeviceInterface, or to extend an existing DeviceInterface through inheritance. When implementing the DeviceInterfaceFactory for an extended DeviceInterface, the inherited methods should not be reimplemented, but should use the result of the DeviceInterfaceFactory for the superclass. Newly introduced methods may be freely implemented, and may as well use the superclass methods. This is required for consistency reasons: Otherwise changes to the superclass behavior may not apply to users of the subclass. If however the behavior of a DeviceInterface shall be changed for a given Device, a **DeviceInterfaceModifier** comes into play. Its implementation is called with

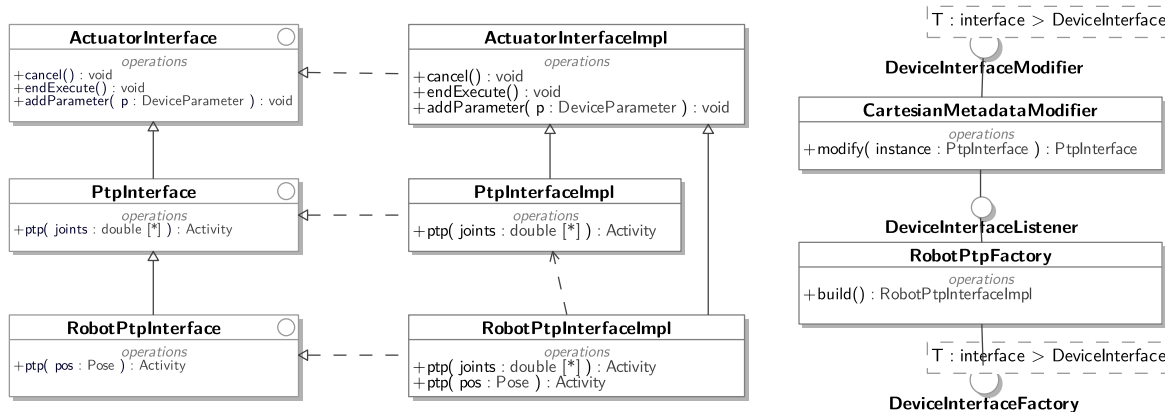


Figure 9.3: Example for DeviceInterfaces for robot arms

an instance of the corresponding `DeviceInterface` and is requested to return a new or modified instance of this `DeviceInterface`. This can be helpful for specialized Devices where new features are added, while the standard features from a more generic Device are to be provided. Generally speaking, different `DeviceInterfaceFactory` implementations can be used to provide the same `DeviceInterface` for different `DeviceDrivers`, depending on the best implementation choice for the corresponding Device and available features of the execution environment.

Figure 9.3 gives an example how the concepts related to `DeviceInterfaces` can be used. As a basis, `PtpInterface` is used as a `DeviceInterface` for Actuators consisting of multiple Joints. It provides access to a trajectory motion in configuration space, where the desired goal position is given as a `double[]`. `PtpInterfaceImpl` as an implementation of this interface is provided by a corresponding `DeviceInterfaceFactory` (not shown in the picture) that is added to Actuators once a compatible `ActuatorDriver` is assigned. To provide this capability for robot arms that can additionally work in Cartesian space, two extensions are shown: The `CartesianMetadataModifier` as a `DeviceInterfaceModifier` augments the `PtpInterface` implementation by Cartesian metadata, e.g. describing the Cartesian position that belongs to the Joint configuration given as a goal of the motion. To further support configuration space motions to a position defined in Cartesian space, the `RobotPtpInterface` extends the `PtpInterface` by a corresponding method. Its implementation – the `RobotPtpInterfaceImpl` – thus has to implement both the method accepting a goal configuration and a Cartesian Pose. Following the implementation guidelines given above, it delegates requests to the configuration space variant to the direct implementation of `PtpInterface`, while it uses the inverse kinematics function of the robot to calculate the goal configuration for the given Pose in the Cartesian case. Instances of `RobotPtpInterfaceImpl` are created by the `RobotPtpFactory` that serves as a `DeviceInterfaceFactory` for the `RobotPtpInterface` as well as a `DeviceInterfaceListener` that waits for `PtpInterfaces` to decide on which Devices it can be used.

Further uses of `DeviceInterfaceModifiers` are robot arms that support the specification of

controller parameters, e.g. the joint impedance of a youBot arm, while still supporting all the motions defined for generic robot arms. There, a `DeviceInterfaceModifier` can extend the motion by a parallel step adapting the controller parameter or an Assignment added to motion Commands. Furthermore, for Cartesian motions a `DeviceInterfaceModifier` can be used to add a step that holds the robot at the specified Cartesian position after the motion has finished, thus allowing to define motions relative to moving coordinate systems.

### 9.1.2 Working with Multiple Execution Environments

Talking about cooperating mobile robots, an application often controls Devices that are connected to different computers and thus execution environments. In this case, multiple `RoboticsRuntimes` are defined, together with `DeviceDrivers` that link the Devices to their physical counterparts. However, it has to be noted that data provided by the `DeviceDriver` of one `RoboticsRuntime` can only be used with tasks executed on the same `RoboticsRuntime`. Likewise, synchronized starting or stopping of tasks may only refer to sensor data and tasks on the same `RoboticsRuntime`. If data about the same Device are required in multiple `RoboticsRuntimes`, multiple `DeviceDrivers` have to be created for the Device, one for each `RoboticsRuntime`. In this context, a distinction about the Device's role can be made (as introduced in [88]): A Device that is connected to a `RoboticsRuntime` where it can be controlled is called a *controlled device*. In another `RoboticsRuntime`, the Device may only provide read-only access with limited timing guarantees. Here it is called a *remote device*. In further cases, no direct access to internal data of the Device may be present, and its state can only be estimated through observation. In this context, a Device can be seen as an *observed device*. For the different roles of Devices, specialized `DeviceDrivers` are available that provide the corresponding access to device data.

Typically, an Actuator can only be controlled in the execution environment it is connected to, and thus only is a *controlled device* in this one `RoboticsRuntime`. The Device's data can additionally be made available for use in other `RoboticsRuntimes` as *remote devices*. This data transfer can either be performed directly between the execution environments, or through an application that monitors the values from the *controlled device* and appropriately forwards it to the *remote devices*. Using *remote devices* can be seen as a step towards a system spanning the two `RoboticsRuntimes`, however in a more explicit fashion.

If a Device is controlled in another application, and its `RoboticsRuntime` is not available in the current application, the best option is to use it as an *observed device*. Here, any kind of sensing that can estimate the state of the Device is applicable. For example, a Vicon tracking system can be used to find the absolute position of a youBot platform, or a laser scanner mounted at one youBot platform allows to estimate the relative position of another youBot platform. However, it may be impossible to retrieve certain internal data of the Device. For a youBot platform, it is impossible to observe the position of its *odometry origin*, i.e. the distance it believes to have traveled since its start. Likewise, torques applied to a youBot arm, or the position set-point of

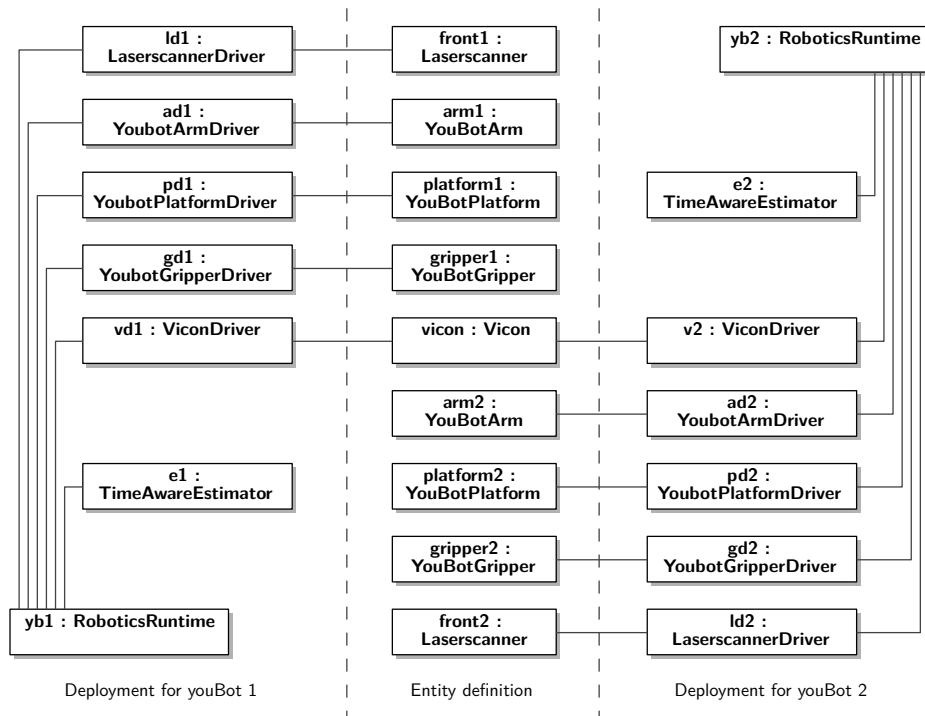


Figure 9.4: Example deployment when controlling two youBots with Vicon tracking

its impedance controller cannot be observed through optical means. In summary, the amount and quality of available information decreases from *controlled devices* over *remote devices* to *observed devices*.

When multiple DeviceDrivers for the same Device are defined in an application, an appropriate choice has to be taken which one to use for a given task. When issuing a task to an Actuator, its role as *controlled device* is chosen, as this is the only one that provides write access to the Device. In contrast, accessing sensor data is also possible for *remote* and sometimes (for geometric data) even *observed devices* through their respective DeviceDrivers. So it depends on the purpose of the sensor data: If it is to be used to control an Actuator, the RoboticsRuntime of its controlling DeviceDriver has to be used to make the sensor data usable in this context. Otherwise, an arbitrary DeviceDriver can be chosen, as all should represent the same knowledge; however the *controlled device* is closest to the hardware device, so if available its data is preferred.

When controlling two youBots from one application and using Vicon to track positions (cf. figure 9.4 in conjunction with figure 8.6), each youBot has its own RoboticsRuntime, controlling the sensors, arm and platform mounted to the youBot. However, for the one Device *vicon* defined in the application geometry, two DeviceDrivers are configured, one for each youBot's RoboticsRuntime. This way, tracking data is available in both RoboticsRuntimes and can thus be used by both youBots for planning or executing their motions. The deployment used here only uses *controlled devices* for the youBots, but no explicit DeviceDriver for *remote* or *observed devices*.

Thus, the `RoboticsRuntime` for the first `youBot` has no information about the configuration of the current Pose of the second `youBot` arm, while at least a position estimate for the `youBot` platform is available through the `ViconObservation`. When the position of the arm is additionally needed, a `DeviceDriver` for a *remote device* can be used in this context.

When using multiple `RoboticsRuntimes`, one more detail of the World model becomes relevant: In the application geometry, aspects such as `Frames` and `PhysicalObjects` have been defined independent from the execution environment, however some `Relations` were still uncertain or unknown. Adding a `DeviceDriver` introduces knowledge about these `Relations` that is specific to its `RoboticsRuntime`. This knowledge is modeled through `Observations` or additional `Relations` added for the given `RoboticsRuntime`, and can be made usable through an `Estimator` (cf. section 8.2). Therefore, each `RoboticsRuntime` gets its own `Estimator` that uses its `RoboticsRuntime`'s view on the World model to create `Relations` based on the `Observations` that are valid for the respective `RoboticsRuntime`. In practice, the structure of `Relations` differs between different `RoboticsRuntime`, leading to different Frame topologies depending on the respective view and knowledge. These created `Relations` use sensor data that describes the Transformation or Velocity, however the Sensor data is only accessible on this `RoboticsRuntime`. When navigating the Frame graph for another `RoboticsRuntime`, e.g. to create a computation law for the desired position, these `Relations` may not be taken into account, whereas application requests to compute the current Transformation between two Frames at a given time may use all `Relations`.

## 9.2 Execution Environment Implementations

As targets of deployment, two different execution environment implementations are available following the proposed approach and implementing the Realtime Primitives Interface. For debugging and simulation, a pure Java implementation (called **Java Control Core**) is provided, as introduced in section 9.2.1. To control hardware devices with real-time guarantees, the Robot Control Core (cf. section 9.2.2) is implemented in C++ for Linux with Xenomai extensions. Furthermore, a Windows version of the Robot Control Core exists that cannot provide real-time guarantees, however provides some basic robot simulation drivers for testing.

### 9.2.1 The Java Control Core – for Debugging and Simulation

The **Java Control Core** provides an implementation of an execution environment for data-flow graphs (cf. section 2.1.1) that runs embedded into an application. It consists of Java implementations of the computation Primitives (as subclasses of `JPrimitive`, e.g. `JBooleanValue`) along with a `JNetExecutor` that evaluates the data flow graphs and handles synchronization rules and simulated device drivers (inheriting from `JDevice`, e.g. `JYoubotArm` and `JVicon`) to handle periphery.



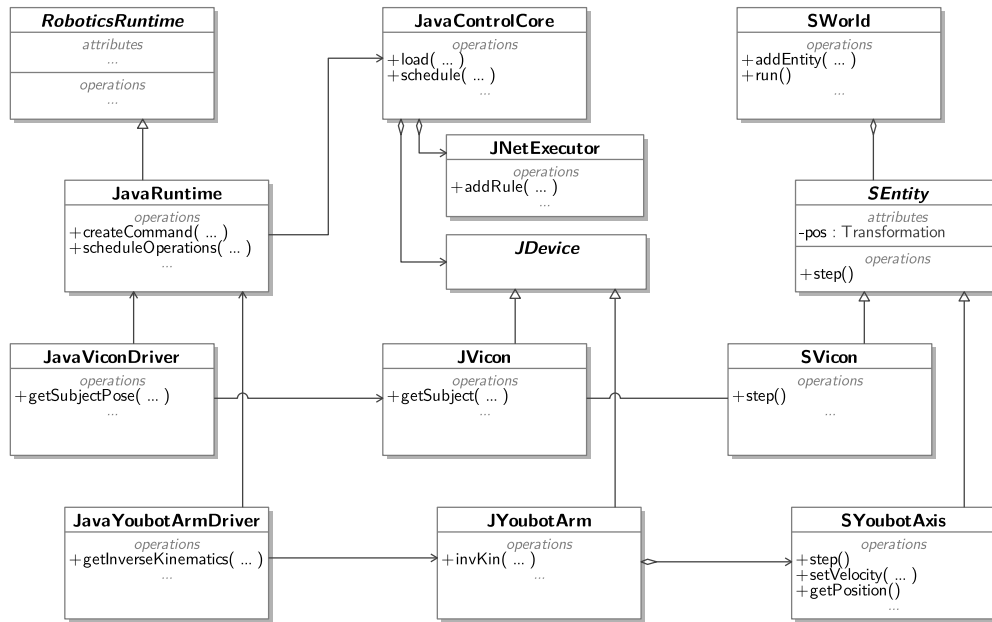


Figure 9.5: Excerpt of classes used with the Java Control Core

The **JavaRuntime** as a **RoboticsRuntime** serves as an entry point to the Java Control Core and can be used with DeviceDrivers such as the *JavaYoubotArmDriver* and *JavaViconDriver* in the application deployment. Issued Commands are transformed into data-flow graphs as introduced in section 6.2, which are then instantiated to create the computation Primitives for execution by the *JNetExecutor*. Being executed within the application, the *JavaRuntime* simplifies debugging, because all debugging features of the development environment (such as break points, stepping and variable inspection) can be used, and object references (and sometimes even stack trace references) exist from incorrectly used Primitives back to the represented Command.

To simulate more complex cases with multiple execution environments, multiple *JavaRuntime* instances can be used within one application. However, in order to work in the same world, some information (especially sensor data) has to be shared, so that e.g. Vicon position data of all tracked *ViconSubjects* is consistently available in all *RoboticsRuntimes*. To achieve this, the simulation of the environment and robot behavior is not included within the Java Control Core devices, but runs independently.

Basis of the simulation is the class **SWorld** with its associated **SEntity**s. An *SEntity* represents a rigid body in the simulated environment that is triggered for periodic time updates and provides its position relative to its parent *SEntity*. This position can be seen as ground truth within the simulation, and can be accessed using simulated sensors with different precision. For each Joint of a youBot arm, an *SYoubotJoint* is used that accepts velocities from the corresponding *JYoubotArm* and computes its position through integration in the time update function. For the Vicon system, the *SVicon* class reads the position of selected simulated *SEntity*s

and provides them to the *JVicon* driver, optionally with measurement noise or time delays.

Figure 9.5 gives an overview over some classes used for a simulated youBot arm with a Vicon tracking system. When using multiple Java Control Cores in one application, the *JVicon* objects can be linked to the same *SVicon* instance, thus working in the same *SWorld* and providing access to the same geometry.

### 9.2.2 The SoftRobot Robot Control Core – with Real-Time Guarantees

While the Java Control Core is well suited for simulation and debugging, it does not provide any real-time guarantees or control for real hardware. In these situations, the **Robot Control Core** that originated from the *SoftRobot* project is applicable (cf. [103]). It is implemented in C++ for Linux with Xenomai extensions, and provides determinism and hard real-time guarantees for data-flow graph and synchronization rule execution and device drivers. Structurally, it is similar to the Java Control Core, with device drivers and executors for data-flow graphs, however runs outside the application process and is accessed through network communication.

To use the Robot Control Core in an application, the **SoftRobotRuntime** class provides the required implementation of a *RoboticsRuntime*, while *DeviceDrivers* such as *SoftRobotYoubotArmDriver* and *SoftRobotViconDriver* represent the adapters linking the *Robotics API Devices* to the *RoboticsRuntime*.

Apart from being able to execute tasks specified by an application, the Robot Control Core is responsible for the communication with its controlled hardware. For this purpose, hardware drivers are loaded as separate components within the execution environment, providing an interface that can be accessed by *Primitives* used in *Data-flow graphs*. Through this interface, sensor data can be read and set-points can be sent to the hardware driver. The hardware driver can include a controller to calculate the data to be sent to the device, or directly forward the received set-points, depending on the granularity of set-points and the interface the hardware device offers.

#### KUKA youBot

For the KUKA youBot, the arm and the platform are both connected to an EtherCAT bus (cf. [48]) that can be accessed from the onboard computer. Arm and platform both consist of a set of joints as EtherCAT slaves, that independently allow position, velocity or motor current control. The joints are equipped with relative encoders, so that for the arm, an initial calibration has to be performed. For this calibration, the joints are slowly moved towards their end stops using velocity control, waiting for the motor current to exceed a threshold indicating that the mechanical limit has been reached. For the platform, the exact initial rotation of the wheels is not important, so that they are simply assumed to be at a zero position. For the arm and platform, independent hardware drivers are used. This way, it is possible to control an arm or a platform alone, or a platform with one or two arms, all connected to the same EtherCAT bus.

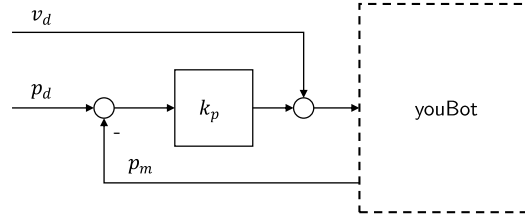


Figure 9.6: Position control loop with velocity feed-forward for the youBot

The platform makes use of the velocity interface of the individual wheel motors, and provides a Cartesian position and velocity interface to Data-flow graphs. It integrates over the measured wheel positions to calculate the estimated position where the platform has moved, and provides this information as well as raw wheel positions and velocities. For position control of the platform, a proportional controller with velocity feed-forward was developed to calculate the desired Cartesian velocities to be applied, using the Cartesian position calculated from the measured wheel positions as feedback. Figure 9.6 shows the general structure of the control loop used, attenuating the difference between the desired position  $p_d$  and the measured position  $p_m$  by the gain factor  $k_p$ , and adding the velocity  $v_d$  of the desired motion as a feed-forward term. The desired Cartesian velocity is limited to the allowed maximum speed and converted into wheel velocities using the kinematics function for the omni-directional platform. The wheel velocities are then forwarded to the wheel joints as velocity set-points.

For the arm, interfaces for position and impedance control on joint level are provided. Position control is implemented using a proportional controller with velocity feed-forward (cf. figure 9.6), providing velocity set-points for the individual joints. Joint impedance control however is implemented on the level of motor currents. For a given goal position, the position error is calculated and converted into a torque based on the configured stiffness and damping that is to be applied. Additionally, the dynamics of the robot require a certain amount of torque to compensate for gravity and inertia. These torques are calculated using a recursive Newton Euler inverse dynamics solver (cf. [27, p. 96]) available in the OROCOS KDL library [95], using the youBot arm dynamics offered on the youBot website [63] and the configured additional load of grasped objects. The resulting torque is then converted into a motor current, based on the torque constant and gear ratio of the motors, and applied as a current set-point for the joint.

In contrast to the youBot arm and platform, the gripper is not connected to the EtherCAT bus as a slave, but to the last arm slave using a serial protocol. Thus, the gripper cannot be controlled in real-time, but can only be given new target positions when the previous gripper motion has completed.

To allow working in Cartesian space, position kinematics functions are provided. The inverse kinematics function for the arm calculates a Joint configuration that brings the end effector to a given Pose. Additionally, it accepts another configuration as hint joints, and chooses the Joint configuration that is closest to these hint joints. However, it is implemented in a strict way, i.e.

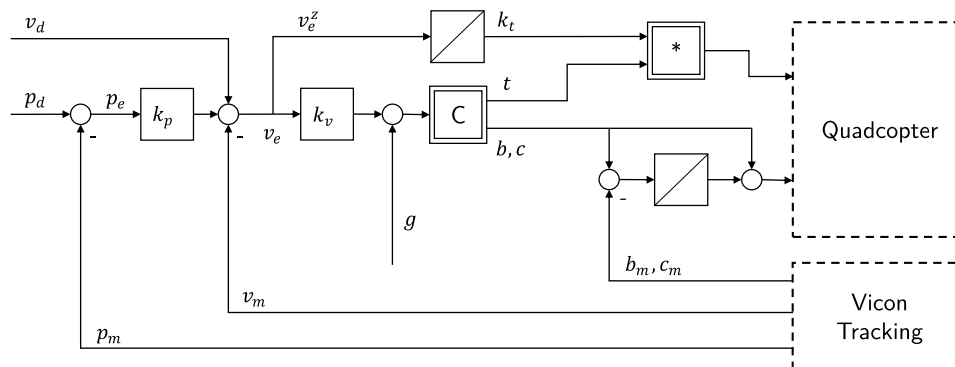


Figure 9.7: Control loop used for position control of the quadcopter

it fails if a position is not reachable. This can happen quite often, because for a given position in space, the Z rotation is restricted, because the X-Y projection of the flange or gripper always has to be parallel to the direction the arm is pointing. To get feasible positions to be used with the kinematics function, a `PositionPreservingProjector` or `OrientationPreservingProjector` is usually applied to correct a Pose so that it becomes reachable (cf. section 5.2.4).

### Autoquad Quadcopter

The quadcopter driver is implemented in a black-box fashion, only accepting attitude and thrust set-points, but not providing any access to internal sensors. It acts as a remote control for the quadcopter, and thus uses an USB DSM2 radio transmitter module, sending values for seven analog channels to the quadcopter. The quadcopter is configured to use manual mode, i.e. to handle the yaw, pitch, roll and thrust channels as raw set-points, interpreting pitch and roll as inclination, yaw as a rotation rate, and thrust as percentage of the maximum thrust, without using GPS navigation or height stabilization through the barometer.

To use the quadcopter, additional sensing has to be configured in the application that provides position and velocity feedback, e.g. through Vicon tracking. The `ActuatorDriver` used in the application is able to create Data-flow graphs that provide position or velocity control of the quadcopter. These Data-flow graphs describe a cascaded control loop (cf. figure 9.7). The outer control loop is handled with a proportional controller with gain  $k_p$  and velocity feed-forward that calculates a desired velocity for the quadcopter, based on the current position  $p_m$  and goal trajectory (with position  $p_d$  and velocity  $v_d$ ), as well as the desired yaw rate (not shown in the picture). Inside, a proportional controller with gain  $k_v$  is used to convert the desired translational velocity into an acceleration, based on the currently measured velocity  $v_m$ . The acceleration is then corrected by adding the gravity vector  $g$ , and a non-linear term  $C$  is used to convert it into pitch  $b$ , roll  $c$  and thrust  $t$ . As the quadcopter has pitch and roll bias and thus is not parallel to the ground if commanded a zero angle, integral terms are used in the velocity control loop based on the pitch and roll angles. Additionally, the factor  $k_t$  required to calculate

the relative thrust for a given acceleration magnitude depends on the weight of the quadcopter as well as the charge of the battery. Here, again an integral term is used to determine  $k_t$  based on the altitude velocity error  $v_e^z$ , which is then multiplied with the desired absolute thrust  $t$ . These values are then sent to the quadcopter to achieve position control and execute the desired trajectory.

### Hokuyo Laser Scanner

For the Hokuyo laser scanner driver, an implementation of the *SCIP2.0* [42] protocol is used. It provides an interface to access the minimum and maximum scan angle, the number of points per scan, and the measurement data and time stamp of the last scan. In the measurement data, invalid readings are expressed as *NaN* values.

To work with laser scanner data in Data-flow graphs, multiple computation primitives are available. One primitive provides the current scan data as a *DoubleArray*, together with a time stamp. Further primitives allow to extract subsets of the scan for a given angle interval, or to combine multiple scans taken from the same position into one scan. Additionally, it is possible to retrieve the measured distance for a given angle from a scan, i.e. a value from the data array.

### Vicon Tracking System

To access the Vicon tracking system, the *Vicon Datastream SDK* is used. The driver provides access to a list of known *ViconSubjects*, a list of unlabeled markers detected in the scene (both without real-time guarantees), as well as position data in (soft) real-time. Position data can be retrieved for a named *ViconSubject*, including the sensed Orientation and a timestamp when the subject has been seen. Additionally, position data for an unlabeled marker can be retrieved. In this case, a position has to be provided for which the closest unlabeled marker is returned. Again, a timestamp is included when the marker has been seen.

As the youBots communicate with the Vicon system through wireless LAN, data packets containing tracking information can be dropped or delayed. Evaluating the actual performance showed that for the arrived data, a certain amount of jitter in the form of time delays up to 8 ms could be observed, while some packets were lost and some arrived delayed by a greater amount of time, but in fast succession to catch up with time. However, all packets contain an increasing frame counter that allows to distinguish between a delayed and a lost packet. Figure 9.8(a) gives an example how packets typically arrive in the given setting. The x-coordinate gives the frame counter, while the y-coordinate represents the time in milliseconds when the packet arrives. The dashed line gives the correct time to which the frame data actually belongs, while the crosses give the time when the data is received.

In this context, the Vicon driver has to estimate a time stamp when received data has actually been measured. As a limitation, the corresponding algorithm has to work with constant time (per data point) and space requirements, and has to allow online computation with real-time

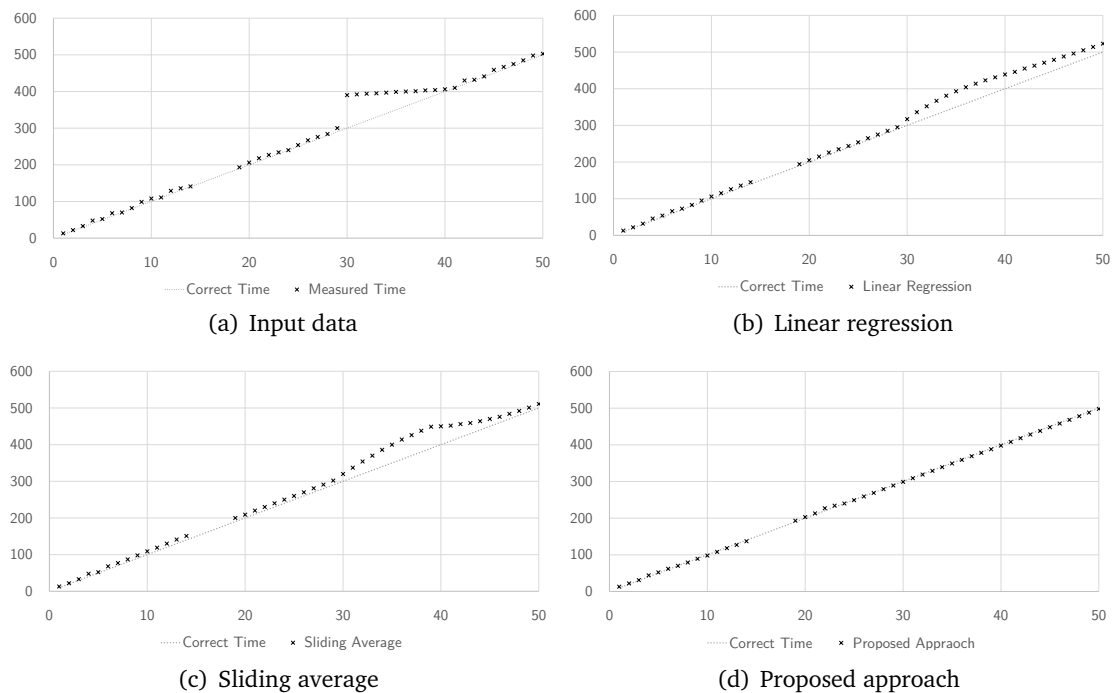


Figure 9.8: Different ways of handling time delays introduced by wireless communication

guarantees. Simple approaches include to take the time of arrival, which however introduces great timing errors in the presence of time delays (cf. figure 9.8(a)), or to perform a linear regression over the frame numbers and arrival times which is then evaluated for the received frame number (cf. figure 9.8(b)). However, as data can only arrive too late but never too early, after time delays the regression will always lead to times that are too big. Another approach is to use a sliding average over the durations between receiving two data points, as suggested by Vistein [103]. However, in the presence of time delays this approach also leads to deviations from the correct time (cf. figure 9.8(c)).

To improve this specific issue, a new algorithm is used to find the correct time. Assuming that measurement data is sampled at a fixed rate and that data can only be delayed, but never arrive before sampled, an optimal estimation would be a straight line (approximating the correct time), which is below each of the sampled data points and minimizes the overall distance. However, as constant time and space requirements do not allow to store all data points to calculate the optimal line, a heuristic approach is used. The approach assumes that the optimal estimation is always defined through two of the data points.

To compute the best approximation, the algorithm works on five data points (cf. figure 9.9,  $p_0$  to  $p_4$ ). The first point  $p_0$  is the first point processed, and thus gives the beginning of the interesting time interval. Similarly, the point  $p_4$  is the last point processed (i.e. the current point), and defines the ending of the interval. The points  $p_1$  and  $p_2$  describe the selected points defining the currently chosen approximation ( $e_1$ ), while  $p_2$  and  $p_3$  represent an alternative

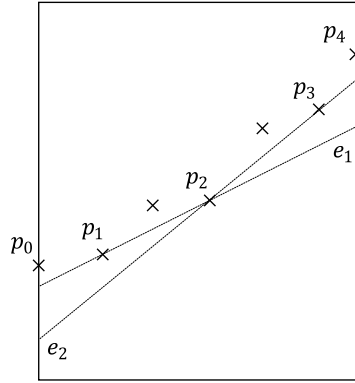


Figure 9.9: Data points and approximations used in the proposed time-smoothing process

approximation ( $e_2$ ).

When a new point is processed, it is first checked against the chosen approximation  $e_1$ . If the new point is lower, it is chosen as  $p_2$  to define the new approximation  $e_1$ , and the other approximation  $e_2$  along with  $p_3$  is invalidated. Otherwise, the point is checked against  $e_2$ , and stored as  $p_3$  to define a new alternative approximation  $e_2$  if lower. This way, the approximations  $e_1$  and  $e_2$  are lowered and make sure that all points since  $p_1$  (or  $p_2$  respectively) are below the approximation, however the approximation cannot become steeper. This is a problem if many initial points have a flat slope, such as if they belong to a sequence that has been delayed by the network and now arrives in a burst.

Thus, a decision logic is added to switch the approximation from  $e_1$  to a steeper  $e_2$  if  $e_2$  has a lower overall distance to the data points than  $e_1$ . This can be decided by comparing the integral of  $e_1$  between  $p_0$  and  $p_4$  to the integral of  $e_2$ <sup>1</sup>. If  $e_2$  has a greater area, it has a lower distance to the data points, and is thus chosen as new approximation  $e_1$ , while  $e_2$  and  $p_3$  are invalidated.

In the given example, this algorithm leads to satisfying results even in the presence of time delays (cf. figure 9.8(d)). Further evaluation based on 10000 random time sequences of 3000 data points each, following the expected scheme (data arrives with random low delay, is suppressed for a random amount of cycles or is delayed for a random greater amount of time and then arrives in a burst) showed that the proposed algorithm performs better than the compared algorithms (cf. figure 9.8) in terms of deviation from the correct time.

### 9.3 Comparison with Previous Work on the Robotics API

Compared with previous work, the new approach changes the software structure between Devices, DeviceDrivers and RoboticsRuntimes. In the work of Angerer [2], a Device had (and

<sup>1</sup>Actually, the integral between  $e_1$  and the data points has to be compared to the integral between  $e_2$  and the data points; however the integral below the data points is equal in these both terms and can thus be omitted.

referenced) no or one `DeviceDriver`. This way, it was not possible to define two independent `DeviceDrivers` for different `RoboticsRuntimes` to re-use the corresponding application geometry of the Device. Instead, Devices used in different `RoboticsRuntimes` (such as the Vicon tracking system) had to be duplicated for each execution environment. In the case of the Vicon system, this leads to different Frames representing the same (tracked) physical object, requiring to choose the right one for the task at hand, thus complicating application development.

In general, the fact that only one `DeviceDriver` was allowed per Device resulted in separate world models for individual `RoboticsRuntimes`, leading to inconsistencies when topology changes such as grasps were only modeled on one topology but not reflected for the other `RoboticsRuntimes`. By inverting the relationship between Device and `DeviceDriver` so that the `DeviceDriver` references its Device and the Device can have multiple `DeviceDrivers`, this problem is solved: For different `RoboticsRuntimes`, a Device can now have individual `DeviceDrivers`, that may additionally have different capabilities: Some may provide read-only data about some aspects of the Device (such as drivers for observed or remote devices), while others allow write access to control the Device. Additionally, it is even possible to use a Device without a `DeviceDriver`, allowing to access its static geometry information (cf. design goals 4.5 and 9.3).

This change additionally allows to change the way of `DeviceInterface` association: While in previous work `DeviceInterfaces` were added based on Devices, the decision to add a `DeviceInterface` is now bound to the `DeviceDriver`. This way, Devices without `DeviceDriver` (that may appear in a mobile robot application for robots controlled from another system) have no capabilities, while adding `DeviceDrivers` for certain `RoboticsRuntimes` incrementally increases the available feature set. Similarly, `SensorInterfaces` (as introduced in section 7.2) now used to access sensor data are added based on `DeviceDrivers`, and can provide sensor data for different `RoboticsRuntimes`, simplifying the definition of sensor-guided motions in setups with multiple `RoboticsRuntimes`.

Looking at available execution environments, the existing Robot Control Core (as described by Vistein [103]) was complemented by the newly added Java Control Core. While the first is still used for real-time control of real hardware, the latter provides better support for debugging and simulation by running in-process and supporting a shared simulated environment for different `RoboticsRuntimes`, allowing easier debugging of cooperating mobile robots with a realistic deployment definition.

## 9.4 Related Work

In *OROCOS*, the deployment describing the used components and their connections can be defined in two ways: First, an XML description (for the Deployment Component) can be used, describing the used components and connections. As a more flexible option, deployment scripts are possible that procedurally create and configure the components and link their ports. This allows to instantiate similar structures through loops, e.g. when multiple quadcopters as a



swarm are used, without having to duplicate the description.

However, this deployment is limited to a single *OROCOS* instance (and thus real-time context), and does not provide means to work on distributed systems. Furthermore, no mechanisms are provided to access multiple *OROCOS* instances from a single application. Additionally, no explicit separation between Devices and their controlling drivers exist, but both are typically provided by a single component, violating design goals 9.1 and 9.3.

Looking at *ROS*, the deployment is defined through XML launch files. These describe the nodes to start and their configuration (including port mappings that define the communication between the different components). Devices are separated from driver implementations into different components, e.g. using a *robot\_state\_publisher* to describe the robot geometry and a *youbot\_wrapper* component providing access to the device capabilities. To make launch files and their parts more reusable, the language *xacro* allows to define macros that can be set for each usage of the launch file, thus allowing to spawn multiple instances of the same robot.

Applications are also modeled as *ROS* nodes and thus started through a launch file. Typically, all these nodes are part of a single system, thus having access to all the data present in the system. However, working with multiple *ROS* systems within one application is not natively supported (cf. design goal 9.3). To transfer data between different *ROS masters*, special components such as *multimaster* or *foreign\_relay* [92] have to be used that transfer certain data from one system to another. These components are explicitly configured through deployment, defining which topics should be mirrored to the other *ROS* master under which name. Additionally, *ROS2* aims at simplifying access to multiple masters as one of its new use cases [34].

In the field of simulation environments for robots, various implementations exist. These range from pure physics libraries such as Bullet [19] (or jBullet [49] as a Java port) to full-blown separate simulation applications such as *Gazebo* [33, 60]. These libraries have an increased feature set compared to the simulation introduced in section 9.2.1 by supporting the interaction between rigid bodies (such as collisions and reactions to them), but are also more complex and computationally intensive. However, these solutions could be integrated into the execution environments for the proposed approach if the additional features become important.

*ROS* provides good simulation support with nodes simulating the behavior of actuators and sensors, and *Gazebo* to simulate the 3D environment, physics of objects and calculating the sensor values to be expected in the simulated situation. Additionally, *ROS* allows to record and play back the messages exchanged between the different components (including time stamps), thus allowing to re-process the input data from an earlier run to debug component behavior. This approach is helpful within the distributed system formed by the *ROS* nodes, because debugging all components at the same time to understand and correct their behavior becomes harder with more components used. In comparison, the simulation Java Control Core typically runs within one process, so that it can be debugged directly.

Looking at hardware support for the KUKA youBot, Keiser [54] suggest a method of torque

control of a KUKA youBot arm. Along with individually optimized PID parameters for improved behavior of the motor current controllers, this approach computes the torques required to perform the desired (Cartesian space) motion using the dynamic model of the youBot. Using this method is claimed to improve the position tracking performance of the youBot arm compared to a method using velocity control with position feedback. Similarly, Ruiz [86] takes a look at the performance of the youBot platform to implement haptic teleoperation. There, he identifies the friction of the base – especially at rest or low speeds – as a major problem concerning precision and haptic transparency of the platform, and identified a model to compensated friction effects. These both approaches provide suggestions how to improve the performance of the reference implementation by changing the device driver for youBots within the Robot Control Core. While these changes offer the chance to transparently improve motion precision, they do not have any implications on the higher layers of the software architecture.

## CASE STUDY IMPLEMENTATION AND EVALUATION

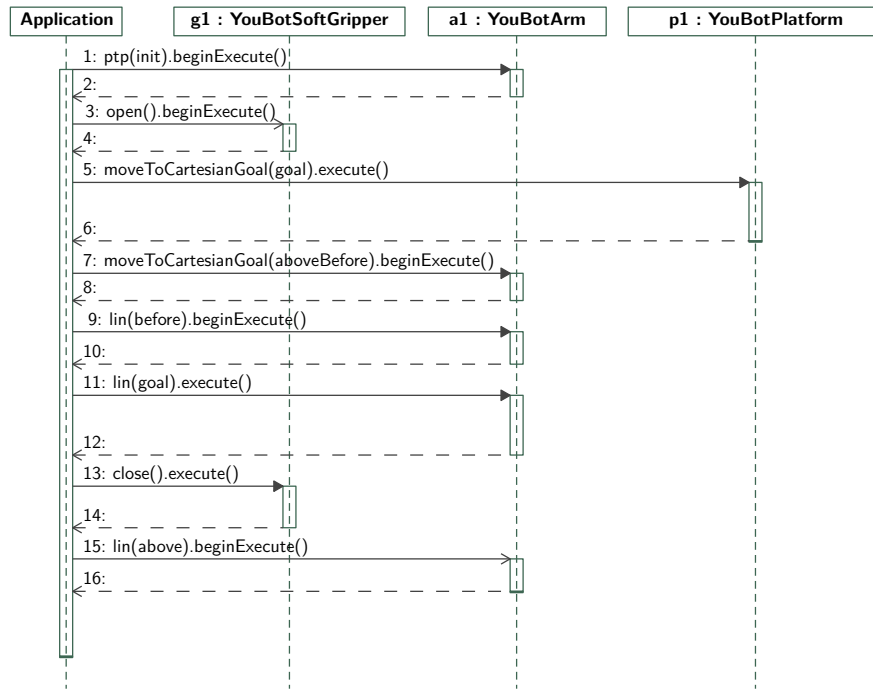
Using the concepts explained in the previous chapters, the application examples introduced in chapter 3 have been realized. The following sections describe details about the implementation of the handover scenario (section 10.1) as well as gesture control (section 10.2), and explain how the different challenges posed in these examples are realized. In section 10.3, the further requirements introduced by aspects of mobility are revisited.

## 10.1 Realizing Handover in Motion

In the handover application, the goal is to pick up a baton from the ground in a predefined zone using the first youBot, and to hand it over to the second youBot while both youBots are in motion. The position of the baton has to be detected using the Hokuyo laser scanner, while geometric synchronization of the two youBots is performed using the Vicon tracking system.

### Application Model

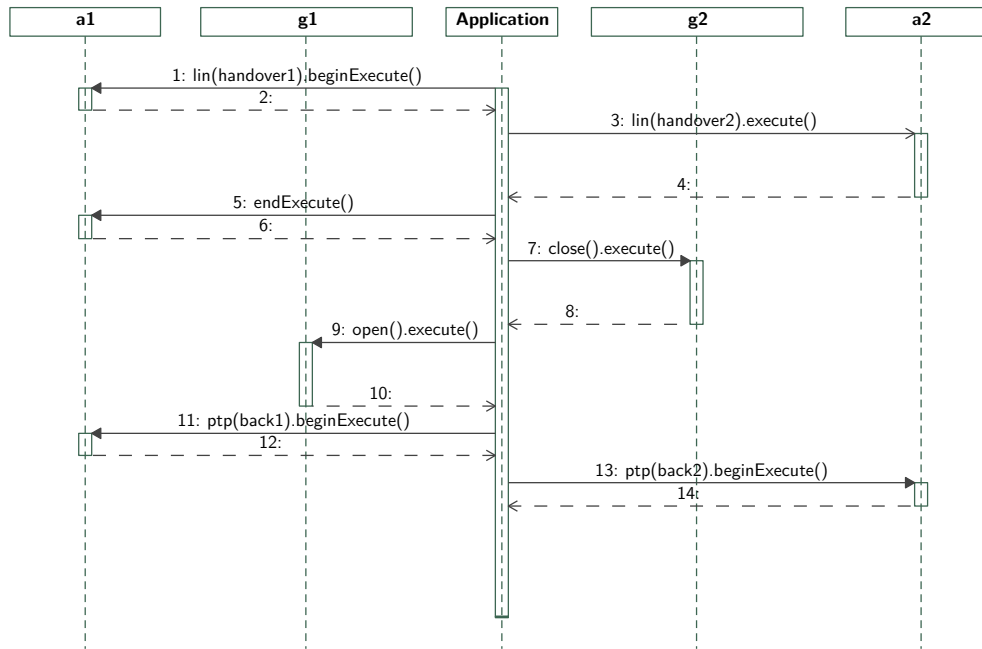
The scenario is implemented in a Java application using the proposed framework. Therefore, the application geometry and used devices are defined through an XML file as described in section 4.3. Based on this geometry, the application is first implemented as one workflow using asynchronous execution of Activitys (using `beginExecute()`) to achieve the required parallelism. For picking up the baton, a `DropSensorListener` is implemented to process the raw distance readings of the laser scanner. The resulting array of *Double* values is first filtered, limiting them to distance readings that are within a radius of 50 cm around the expected pick up position. As the pick up zone is defined relative to the Vicon origin Frame, the laser scanner data is

Figure 10.1: Workflow for picking up the baton using parallelism with `beginExecute()`

augmented by a `RealtimePoint` describing the position of the pick up zone relative to the laser scanner scan Frame through the definition of a `RealtimeTuple`.

After filtering the data, a pole detection algorithm is used. This algorithm iterates over the range measurements and searches for clusters of a length corresponding to the expected diameter of the baton (4 cm). Therefore, for each value the distance to the cluster of previous values is checked, and the cluster is extended if the length is within the allowed diameter, or ended if outside. Whenever a cluster is ended, its length (number of values) is checked against the allowed length for the expected diameter. As the laser scans are represented in polar coordinates, this allowed length depends on the distance of the baton, and can be approximated through the  $\arcsin$  trigonometric function. When a cluster of the expected length is detected, its center point is computed and returned as a result of the pole detection.

Using the detected cluster position and the Transformation of the scan Frame relative to the Vicon origin, the observed position of the baton is updated. This observation and update process is used until the youBot platform has come sufficiently close to the baton, and is then stopped to avoid that the arm follows the measurement noise while picking up the baton. To pick up the baton, a `PositionPreservingProjector` is used with the youBot arm, using the property that the baton is round and thus the exact orientation of the youBot gripper does not matter. First, a position above and before the baton is approached using `moveToCartesianGoal()`, and then a downward and forward motion follows through two `lin()` motions with motion blending (cf. figure 10.1). After closing the gripper, `lin()` is again used to pick up the baton still using the pole

Figure 10.2: Workflow for passing the baton using synchronization with `beginExecute()`

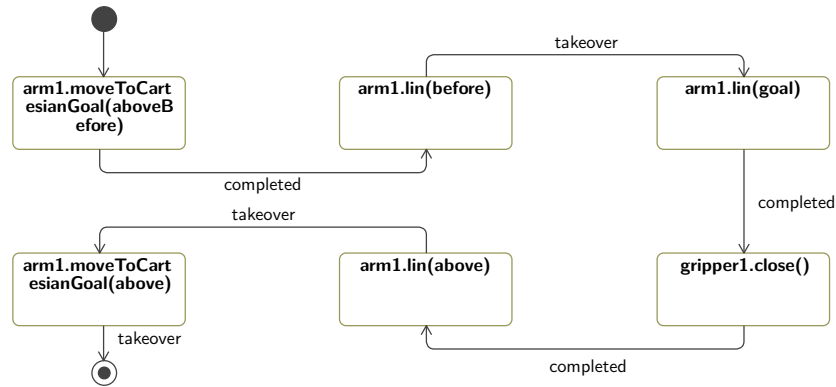
position as a reference Frame, and `moveToCartesianGoal()` allows to return to the coordinate system of the youBot.

In the second phase, the motion of both youBots is synchronized. Therefore, the second youBot platform is commanded to drive next to the first youBot using the `moveToCartesianGoal()` method in the corresponding DeviceInterface after the first youBot has moved to the handover zone and started a slow motion during which the baton is to be transferred.

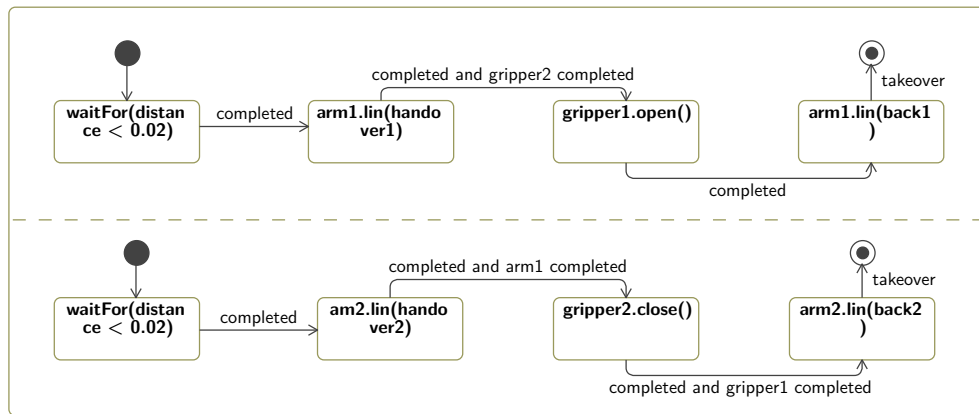
To perform the handover in the third phase, first both youBots move their arms towards each other using `lin()` to a position in the middle between the two youBots, with the second youBot gripper below the first (cf. figure 10.2). Once both grippers arrive at their handover position, the second youBot gripper is closed, and afterwards the first youBot gripper is opened. Once this is completed, both arms retreat, ending the takeover sequence.

Using this approach works, however shows noticeable time delays before and after commanding the grippers when executed on the real robots. This is caused by the use of `execute()` and `endExecute()` for synchronization, because the following commands are only prepared and loaded after the previous Activity has completed. The wireless connection to the youBots used to transfer the data-flow graphs and synchronization rules imposes communication round-trip times and provides only limited network bandwidth, so that typically delay times of 300 ms to 800 ms occurred.

As this happens while both youBot platforms are in motion, it extends the distance that has to be traveled in parallel. To reduce this problem and limit the delay times, the workflow for picking up the baton (cf. figure 10.3(a)) and especially for passing the baton can also be



(a) Sequence for picking up the baton modeled as state chart



(b) Passing the baton with synchronization through state charts

Figure 10.3: Handover workflow with state charts for reduced time delays

implemented by state charts (cf. figure 10.3(b)).

For picking up the baton, the resulting StateChartActivity describes a sequential process, using motion blending in some places. Figure 10.4 shows Java example code how to create the StateChartActivity, using the Java 8 Lambda expressions to allow the concise definition of States as ActivityFactorys. Through *addCompletionTransition()* and *addTakeoverTransition()*, Transitions are created that filter for different types of ActivityResults and result in synchronization (by waiting for completion) or continuous motions (using motion blending). Figure 10.3(a) shows a graphical representation of the resulting state chart. For passing the baton, two StateChartActivities are created, one for each youBot arm (together with its gripper), and started in parallel using *beginExecute()*. The StateChartActivities first wait for the platforms (that are commanded separately) to reach the desired position relative to each other, and then move the arm to the handover position using *lin()*. Once this motion is completed for both youBots, the transition for the second youBot to close its gripper is taken. The first youBot however waits for the second

```

1 // accessing the device interfaces ...
2 CartesianGoalMotionInterface armCG =
    arm.use(CartesianGoalMotionInterface.class);
3 LinearMotionInterface armLM = arm.use(LinearMotionInterface.class);
4 GrippingInterface gripperGI = gripper.use(GrippingInterface.class);
5 // defining states ...
6 State moveStart = () -> armCG.moveToCartesianGoal(aboveBefore);
7 State moveBefore = () -> armLM.lin(before);
8 State moveToGoal = () -> armLM.lin(goal);
9 State grasp = () -> gripperGI.close();
10 State moveUp = () -> armCG.moveToCartesianGoal(above);
11 // setting up the state chart ...
12 StateChartActivity sc = new StateChartActivity(moveStart, arm,
    gripper);
13 sc.addCompletionTransition(moveStart, moveBefore);
14 sc.addTakeoverTransition(moveBefore, moveToGoal);
15 sc.addCompletionTransition(moveToGoal, grasp);
16 sc.addCompletionTransition(grasp, moveUp);
17 sc.addTakeoverTransition(moveUp, null);
18 sc.execute();

```

Figure 10.4: Java code to create and execute the StateChartActivity for picking up the baton

gripper to be closed, before leaving the *lin()* state, and opens its gripper afterwards. Once both gripper tasks are completed, the arms are again retreated. These processes are essentially sequences, however a StateChartActivity has been chosen in favor of a SequentialActivity, because the individual tasks are long enough to allow an incremental preparation of the successor states, and this way the tasks can already be started before all steps have been prepared, reducing the time delay before the interaction starts. For synchronization between the two youBots, the state charts react to completion of each other's Activitys. However, as the youBots are not assumed to work in a common system, direct reaction to each other's progress is not possible. Instead, the completion events are forwarded through the application, reacting to the completion of Activitys by changing RealtimeBooleans used in the other state chart. This way, no real-time guarantees are provided for this workflow synchronization, however the amount of data to be transferred to trigger the transition is significantly lower than for loading and starting a new Activity, so the time delays can be reduced below 300 ms.

### Deployment and Execution

To execute the application, the deployment configuration was performed as introduced in section 9.1.2 through a further XML file. For using the real youBots, two Robot Control Cores running on the youBots were used, while the application was executed on a laptop computer. The baton was placed in the expected pick-up zone and the youBots in their respective starting area, and the handover application was started. Multiple experiments showed that the baton

was reliably detected by the laser scanner, and a sufficiently exact position was provided to grasp the baton. Using the synchronization with `beginExecute()`, a relatively long parallel driving area was required, which could be improved by using the state chart implementation. To test the performance of the used `TimeAwareEstimator`, Vicon tracking was once disabled during the handover phase. As the motion of the first youBot had a constant velocity at that time, the `TimeAwareEstimator` succeeded in extrapolating the position, so that the handover could be completed. A video of the handover application can be found at <http://video.isse.de/handover>.

In addition to real-world execution, the handover was also tested in simulation. Using the Windows Robot Control Core, the deployment had to be changed to reflect the reduced simulation support of this solution: Here, only a single Robot Control Core could be used, and no exteroceptive sensors such as Vicon tracking or Hokuyo laser scanners were available. Instead, known Placements were introduced into the World model to cover all aspects usually provided by the Estimator (e.g. the position of the *odometry origins* and of the baton). Using this set-up, the application could be executed, while using the 3D visualization engine available in the framework to see the resulting robot behavior. Additionally, this simulation allowed to find bugs in the implementation or use of computation Primitives within the Robot Control Core, without having to run the test on real hardware.

For a more realistic simulation, two set-ups with the Java Control Core were used. Both use the same application geometry as for the real youBots, and additionally configure the simulation environment by setting up a `SWorld` with `SYoubotPlatforms`, `SYoubotArms`, `SYoubotGrippers`, `SUrg` simulated laser scanners, an `SVicon` instance along with its `SViconEntitys` configuring the tracked *Subjects*, and the baton to be detected using the laser scanner. The first set-up then uses a single Java Control Core instance controlling all the devices, and a `SimpleEstimator` to handle Observations. Executing the application in this set-up also allows to use the 3D visualization engine and behaves similar to the Robot Control Core simulation case, however additionally allows to test sensor processing in the form of Estimators and the pole detection algorithm for the laser scanner. Additionally, through using only one `RoboticsRuntime` and the less complex `SimpleEstimator`, this simulation set-up can also be used on slower computer hardware. To get even closer to reality, the second set-up uses two `JavaRuntimes` for the two youBots and `TimeAwareEstimators`. While more computationally complex, this configuration allows to test the entire application under realistic conditions, including aspects related to the use of multiple `RoboticsRuntimes`. Here, programming errors concerning the use of `RealtimeValues` of the wrong `RoboticsRuntime` or real-time coordination aspects that cannot span different `RoboticsRuntimes` can be found, and additionally the `RoboticsRuntimes` have more realistic (limitations of) data about other Devices (e.g. the fact that the second youBot does not know about the Pose of the first youBot *flange*).

Changing between these different deployments is only a matter of selecting or editing



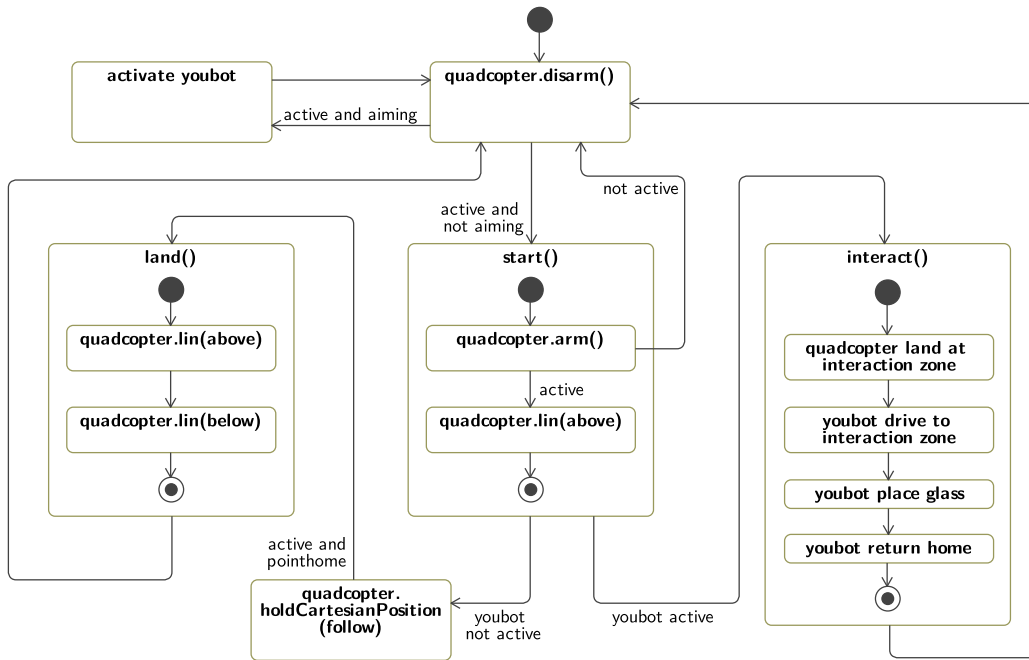


Figure 10.5: Workflow state chart for the gesture control application

an XML configuration file, while the application implementation remains unchanged. The application – here implemented to work with two systems – can thus also be executed using a single *System* without changes, as requested in section 2.3.3.

## 10.2 Realizing Gesture Control of the Quadcopter

In contrast to handover in motion, the gesture control application focuses more on user input and reactive behavior. An operator uses a wand (a wooden stick with Vicon markers) as a pointing device to trigger certain behaviors of a quadcopter, to guide the quadcopter along arbitrary paths and to make a youBot place a filled glass on top of the quadcopter.

### Application Model

The implementation uses an application geometry definition as described in section 4.3, and uses a state machine to realize the expected behavior. Figure 10.5 gives an overview of the main coordination used in the application. Initially, the quadcopter is disarmed, waiting for the wand to become activated. When the wand is activated while pointing at the youBot, the youBot is activated, bringing its arm into an appropriate start position. Otherwise, the quadcopter is started. Therefore, it is first armed, and then flies up 1 m to safely leave the landing zone. If however the wand is no longer active after the quadcopter is armed, the quadcopter is disarmed to return to its waiting state. After flying up, the application checks whether the

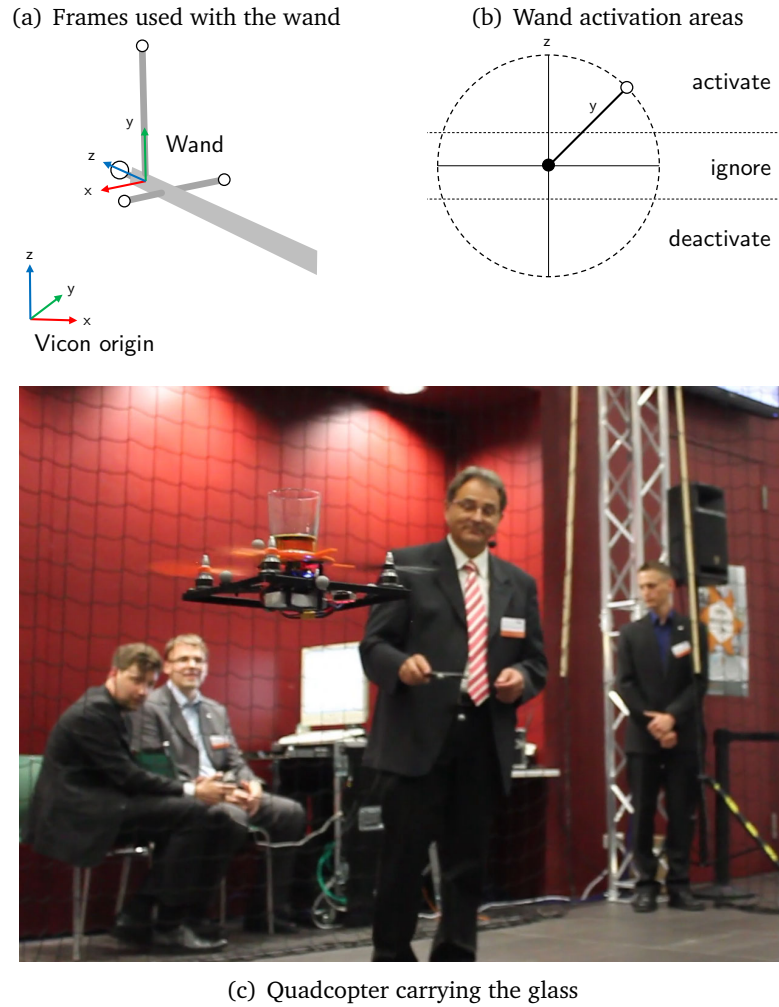


Figure 10.6: Example application: Gesture control

youBot is activated, and either starts the interaction sequence or flies the quadcopter to a goal position 50 cm in front of the wand. This goal position is provided by the application through a *RealtimePoseFromApplication* that is updated through a *DropSensorListener* that observes the *RealtimePose* of the wand tracked by the Vicon system. This indirection through the application is used to implement Pose filtering in the application, rejecting positions that are too close to the landing zone or that occur while the wand is inactive. When during flight the wand is active and its tip is close to the landing zone, the landing sequence is initiated that flies the quadcopter above the landing zone, and then slowly below to effectively land the quadcopter despite the ground effect. Then, the quadcopter is disarmed and back in its waiting state.

For the interaction, after the quadcopter has started it flies to the interaction zone and lands there. Then, the youBot platform drives next to the landing zone, and the arm grasps the glass placed at a known position on the aluminum plate. Then, the arm is moved to a position above

```

1  Vicon vicon = ...
2  ViconSubject wand = ...
3  // Y axis of the wand ...
4  RealtimePose up = new RealtimePose(wand.getFrame(),
    RealtimeTransformation.fromConstant(0, 1, 0));
5  // ... expressed in Vicon Orientation (defining 'up') ...
6  RealtimePose viconUp =
    up.withOrientation(vicon.getOrigin().getOrientation());
7  // ... extracting the Z component ...
8  RealtimeDouble zAmount = viconUp.getTransformation().getZ();
9  // ... above or below zero ...
10 RealtimeBoolean activate = zAmount.greater(0.3);
11 RealtimeBoolean deactivate = zAmount.less(-0.3);

```

Figure 10.7: Java code to determine the wand activation

the quadcopter (using a *ViconObservation* to describe the position of the quadcopter), and the glass is carefully placed on top of the quadcopter. Using the real position through Vicon tracking instead of the commanded landing position here allows the youBot arm to compensate for the landing inaccuracy of the quadcopter as well as for the driving inaccuracy of the youBot platform. To end the sequence, the youBot returns to its start position, while the quadcopter is in its disarmed state and can be started again by activating the wand.

Apart from the state machine for discrete switching, a set of *RealtimeBooleans* describing different situations are defined and used as guards for state machine transitions (*active*, *poonthome*, *youbot active*, *aiming*, and combinations of these). In figure 10.6(a), the wand uses a coordinate system with X pointing left, Y pointing up and Z pointing along the wand. The wand is assumed to be active when its prominent marker points upwards, so activation corresponds with the geometric property that the Y axis of the wand coordinate system points upwards. This property can easily be expressed using the introduced concepts of Poses: A Pose with the wand Frame as reference and a translation of 1 m in Y direction is defined, and then changed to use the *Vicon origin* as Orientation. Whenever the resulting Pose has a positive Z coordinate, the wand is active. Figure 10.7 shows a Java code example that performs this calculation. To make activation detection more robust, a hysteresis is used, activating the wand only if Z is above 30 cm and deactivating it when Z is below –30 cm (cf. figure 10.6(b)).

The second interesting situation is whether the wand tip is close to the landing zone. This can be calculated from the Transformation from the wand Frame to the landing zone Frame by inspecting the length of the corresponding translation. Finally, the third decision is whether the wand points at the youBot. Therefore, the Pose of the youBot relative to the wand Frame is calculated, using the wand Frame as Orientation. Then, the wand points at the youBot if the Z coordinate is positive and the magnitude of X and Y are below a specified threshold, here using 50 cm.

### **Deployment and Execution**

This scenario can again be executed in simulation and using real hardware. In the simulation case, a deployment configuration with one Java Control Core is used, simulating the youBot and quadcopter. In addition to the concepts used in the handover application, a SEntity for the wand had to be implemented that allows changing the position and orientation through a user interface. Using this together with the simulated youBot and quadcopter allowed to run the application, testing the implementation of the quadcopter controller as well as the switching logic of the application.

To run the application with real hardware, two Robot Control Cores are used, one for the youBot and one for a stationary PC controlling the quadcopter. Here, a deployment configuration similar to the one presented in section 9.1.2 is used. This application was presented as a closing demonstration at the “Open Research Lab” 2015 event organized by the Institute for Software and Systems Engineering at the University of Augsburg. Some impressions of this show can be found in the video available at <http://video.isse.de/gesturecontrol> (cf. figure 10.6(c)). Using the proposed framework allowed to successfully implement and test the application within few days, using a common programming approach for the different devices to reach a consistent and reliable application.

## **10.3 Realization of Requirements**

In section 2.2, requirements have been given that are essential to effective object-oriented programming of mobile robots. In chapters 4 to 9, these requirements have been refined to more concrete design goals that guided the design of the proposed software framework. Looking at the result, the requirements have been achieved, as detailed in the following paragraphs.

### **Requirements 1 to 5: Specification of geometry, motions and tasks**

These requirements have been derived from the work of Angerer [2], and were already fulfilled for industrial robots in the solution proposed there. Being based on this object-oriented model, the requirements are still fulfilled in the work proposed in this thesis. Additionally, some facets of these requirements that become especially important with mobile and distributed robots have been included as design goals and integrated into the proposed approach. This especially includes the introduction of light-weight geometric data types to simplify the specification of positions in space and avoid programming errors (cf. section 4.1.4), the handling of kinematically redundant robots through FrameProjectors (cf. section 5.2.4) and the separation of composed tasks into individual data-flow graphs, reducing computation overhead and allowing more flexible reactions (cf. section 6.2). Additionally, the possibility to define the possible outcomes of tasks has been added, allowing mobile robots to cope with unexpected situations and flexibly react to them in real-time.

**Requirement 6: Flexible integration of sensor data into the world model**

Through the introduction of unknown or uncertain Relations, Observations and Estimators, a flexible specification mechanism has been developed that allows to define the connection between sensor data and geometric aspects modeled in the world model (cf. section 8.2). This allows to handle a key aspect of mobile robots – their intrinsic inaccuracy compared to robot arms – in a way so that application programming can remain similar to the approach introduced by Angerer [2] for industrial robots.

**Requirement 7: Support for reactive behavior**

With the new StateChartActivity and ThreadActivity (cf. section 7.3), the support for reactive behavior in mobile robot applications has largely been extended, solving the problem of long preparation times and huge data-flow graphs that occurred for task sequences with previous SequentialActivitys. Additionally, they allow to consistently handle case distinctions and system reactions (that are often encountered by mobile robots in less structured environments), while encapsulating the resulting tasks as Activitys that can further be composed and used like other Activitys. For example in mobile robotics, these Activitys make it possible to plan next execution steps incrementally or up front, handling the different possible outcomes of executed tasks.

**Requirement 8: Support for devices connected to different computers**

The solution proposed in this thesis puts great emphasis on the separation between application programming and deployment definition. Devices used in an application may belong to a single RoboticsRuntime, but also to different RoboticsRuntimes for separate robots. This is especially relevant for mobile platforms such as the youBot that do not have a deterministic network connection and thus cannot be included into a common real-time context. Unless specifying real-time reactions, it makes no difference to applications if two Devices are connected to the same RoboticsRuntime, if they have individual ones or even if one Device is known in two RoboticsRuntimes. The Devices can still be accessed consistently, working on a shared world model. To achieve this, internal concepts such as FrameTopologys (cf. section 4.1.2) but also Estimators (section 8.2.2) play an important role, allowing to abstract from the changes and access data in a consistent way.

In summary, the basic requirements for a software framework supporting mobile robots have been fulfilled. Based on these foundations, it is now possible to implement further capabilities of mobile robots, such as mapping, navigation and collision-free motion planning, but also high-level task planning and learning algorithms, while making use of existing libraries as well as development tools for object-oriented software development.



## CONCLUSION

This work aimed at developing and evaluating an approach that allows the object-oriented modeling and coordination of mobile robots, supporting the cooperation of mobile manipulators. Based on the developed concepts, two novel and challenging example applications were realized and tested, featuring the physical interaction of two youBots as well as the cooperation of a youBot with a quadcopter and manual guiding and gesture control of a quadcopter. Using the software framework presented in this thesis, the corresponding applications could be implemented with low effort, focusing on the task description and an object-oriented model of the problem domain. This allowed to rapidly implement the quadcopter application, emphasizing the capability and efficiency of the proposed approach. The further aspects required before an application controlling hardware can actually be executed – ranging from low-level device control to task scheduling, data processing and distribution – were handled by the framework, allowing the developer to focus on the important, application-specific aspects. To facilitate this, a flexible and extensible approach has been presented as an important extension to the work of Angerer [2], Hoffmann [39], and Vistein [103], that allows object-oriented modeling and coordination of mobile robots.

In chapter 4, a powerful way of describing geometric aspects has been introduced, with expressive concepts allowing to describe positions and velocities in the way most suitable for the desired application. Additionally, a way of modeling the structure of physical objects and Devices has been described that forms an important prerequisite to a deployment-agnostic geometry description. It allows to describe the set-up independent of the decision which execution environments the Actuators will be connected to, and supports working with multiple execution environments in a single application. In this context, uncertain and unknown Relations have been introduced to model logical relationships for which no geometric data is known yet, which will however be filled during application run-time through sensing.

Next, an extensive model of motions and device Actions has been introduced in chapter 5, extending the range from industrial robot motions that are mainly defined for static environments to more flexible motions, and introducing means of classifying these Actions into different categories. This motion model allows to specify motions independent from the Actuator that will execute them, and is equally applicable to robot arms as well as mobile platforms and even quadcopters. Furthermore, parameters to configure motion execution have been described for giving motion and speed limits, defining the relevant position of an Actuator for which the motion is given, and handling the kinematic restrictions of robots – both arms and platforms – with less than six degrees of freedom through *FrameProjectors*. The latter form a vital prerequisite to making the motion definitions independent from Actuators, allowing to define arbitrary motions in Cartesian space and to independently configure which aspects of the motion are most important for execution on a given Actuator and should thus be prioritized.

Using these Actions, a model for Commands was introduced in chapter 6, allowing to define tasks to be executed by Actuators, to react to situations that occur during execution with timing guarantees, and to define further continuous processes that should be performed synchronized with the main task executed. Therefore, a model of run-time data described by *RealtimeValues* has been introduced to consistently access time-variable data within Commands and independently, simplifying the definition of Commands as well as the implementation of Actions for describing continuous processes. These Commands support different outcomes that are explicitly modeled as *CommandResults* and can be reacted to in real-time to facilitate case distinctions, supporting mobile robot reactions to their environment. Moreover, a transformation process from Commands to executable data-flow graphs has been introduced, removing unnecessary complications of the previous approaches and mapping to the powerful concept of synchronization rules as introduced by Vistein [103], solving the limitations of the previous approach by Angerer [2] concerning the concurrent control of multiple devices and its transition to new tasks.

In chapter 7, the different levels of robot synchronization have been introduced, along with ways how to achieve them in the proposed framework. Device capabilities are modeled as *Activitys* that provide descriptions about their expected outcome and thus allow to plan the following steps based on this metadata. These *Activitys* can be composed to create more complex capabilities, allowing both transitions with real-time guarantees and expressive reactive behavior as often required for mobile robots in less structured environments. These compositions are *Activitys* themselves, thus allowing further combination. The possible *Activitys* are added to Devices in an extensible and consistent way, allowing to introduce new Device features based on existing features at run-time, and also to provide complex reactive behavior composed of other *Activitys* as native Device features. For execution, *Activitys* rely on the Command mechanisms, inheriting their advantages while simplifying application programming by internally planning sequences based on predecessor metadata.

Working in less structured environments, processing of sensor data is important for mobile



---

robots. Thus, chapter 8 introduces ways to access sensor data – with significant extensions to previous work when handling sensor data that is not always available immediately, allowing to handle sensors with lower update rate or longer processing times by providing time consistent sets of values. Additionally, the introduced concepts of Observations and Estimators as an important and novel approach presented in this work allow to independently define the relationship between sensor data and geometric aspects in the environment, forming the basis for filling the gaps of unknown Relations defined for the application geometry. This way, sensor processing can be handled as a deployment issue, separated from the defined application workflow and allowing reuse in other environments using different types of sensing.

Finally, deployment and execution issues have been described in chapter 9. There, a way of connecting the Devices defined in the application geometry to real hardware devices is introduced, allowing to move deployment decisions to a late stage of application development by keeping the application independent from these details. Furthermore, this approach allows to represent single Devices on multiple execution environments, achieving a common world model in the application despite different knowledge in the individual execution environments. This common world model serves as a central place for describing the environment, where changes applied by one Actuator are automatically available to other Devices working in the same world model. Moreover, two execution environment implementations have been introduced, one based on the work of Vistein [103] that supports the hardware used in the case study and provides real-time guarantees, and one implemented in pure Java for simulation and debugging.

The suggested framework maintains the positive aspects of the earlier work and extends them to consistently handle the issues of geometric uncertainty vital for mobile robots used in less structured environments, and distribution as required for the cooperation of mobile robots. It offers compositionality on three levels when combining a mobile platform with a robot arm into a mobile manipulator: For kinematics through the combination of individual FrameProjectors, through synchronized Commands for the individual Devices to achieve common tasks with real-time guarantees, and for tasks through composed Activitys encapsulating complex device capabilities that can be provided and further combined. In this respect, the proposed approach has major advantages in comparison with popular component frameworks for robotics used today, by providing real-time guarantees for composition of tasks of the individual components of a mobile manipulator, while allowing to specify the individual parts independently and without the usual restrictions of real-time programming.

The approach addresses important topics with a focus on *Industry 4.0*, where greater flexibility of production can be achieved through the introduction of mobility. Additionally, it harmonizes motion definition for different types of robots, improving the exchangeability of Actuators with minor modifications to the application definition. This focus also becomes visible in current work on new ISO standards, especially concerning “Modularity for industrial and service robots”, trying to “establish a common base for compatibility between hard- and

software components from different manufacturers” [46].

The results of this thesis are planned to be used as foundations for further research. In the field of self-organizing robot swarms, the framework will provide object-oriented means of controlling quadcopters, bridging the gap between an agent system deciding about the desired behavior and low-level hardware control. Therefore, the quadcopters will carry an onboard computer running a control core, an instance of the object-oriented software framework as well as an agent framework for behavior planning and decision making. The interaction between agent and robot framework will then allow to define and implement swarm behavior through the coordinated execution of motions and tasks. Workflow synchronization will be implemented on the agent level, while (weak forms of) time synchronization are required for achieving the desired flight patterns of the quadcopter swarm.

Furthermore, the cooperation of large teams of industrial robots will be one focus of a cooperation between the Institute for Software and Systems Engineering and the DLR Center for Lightweight Production Technology. There, a huge robot cell with multiple robot arms mounted on linear units is to be programmed. The cell is intended for the production of airplane and spaceship parts, requiring the large dimensions as well as the cooperation of multiple robots to handle large components. In this context, support for distribution on the real-time level has to be established to avoid having to perform all real-time control centrally. Therefore, the presented concepts towards real-time distribution – transforming Command structures into individual data-flow graphs with synchronization rules – play an important role for defining places of distribution and keeping the data dependencies manageable. This way, some aspects developed for mobile robots in this thesis will be applied to industrial robots, shifting back the focus while keeping the advantages of distribution.

Moreover, the topic of constraint-based programming of robot motions will be further addressed. Especially in the field of mobile manipulators, it promises to make even greater use of the redundancy present in these systems, improving compositionality beyond the level reached in the proposed approach. For example, for a two-arm manipulator a bi-manual manipulation task can be described with a focus on the task (e.g. that the bottle cap should be rotated against the bottle), while keeping the less important aspects (e.g. where the bottle is held in the work space) open to be controlled from lower-priority tasks. In this context, the introduced approach provides a helpful environment for implementing such behavior, using domain modeling as well as means of real-time execution.

Further plans exist to improve interoperability with other robot frameworks (especially *ROS*), allowing to use the existing algorithm implementations where real-time is less important, and to combine the strengths of different approaches. In this context, it is important to note that many basic concepts introduced in this thesis are not tied to a specific framework, but instead very general and can be implemented in different frameworks, providing a way to greatly simplify the construction of software for mobile robot systems.

## BIBLIOGRAPHY

- [1] ABB Robotics. *Operating manual–introduction to Rapid*. 2004 (cit. on p. 7).
- [2] A. Angerer. “Object-oriented Software for Industrial Robots”. PhD thesis. University of Augsburg, Mar. 2014 (cit. on pp. 1, 2, 7, 8, 10, 16, 18, 23, 41, 64, 65, 70, 85, 104, 105, 107, 112, 114, 115, 128, 129, 147, 165, 178, 179, 181, 182).
- [3] A. Angerer, A. Hoffmann, A. Schierl, M. Vistein, and W. Reif. “Robotics API: Object-Oriented Software Development for Industrial Robots”. In: *Journal of Software Engineering for Robotics* Vol. 4.1 (2013), pp. 1–22 (cit. on pp. 112, 128).
- [4] B. Arulampalam. *Beyond the Kalman Filter: Particle Filters for Tracking Applications*. Artech House, 2004. ISBN: 9781580538510 (cit. on p. 148).
- [5] *AutoQuad M4 – Micro Flight Controller*. Online, accessed Feb 2016. URL: <http://autoquad.org/wiki/wiki/m4-microcontroller/> (cit. on p. 32).
- [6] M. Barnes and E. L. Finch. *COLLADA - Digital Asset Schema Release 1.5.0*. Online, accessed Feb 2016. The Khronos Group. 2008. URL: [https://www.khronos.org/files/collada%5C\\_spec%5C\\_1%5C\\_5.pdf](https://www.khronos.org/files/collada%5C_spec%5C_1%5C_5.pdf) (cit. on p. 67).
- [7] *BionicTripod with FinGripper – versatile movement and adaptive grasping*. Online, accessed Feb 2016. URL: [https://www.festo.com/cms/de\\_corp/9779.htm](https://www.festo.com/cms/de_corp/9779.htm) (cit. on p. 32).
- [8] R. H. Bishop. *Modern Control Systems Analysis and Design Using MATLAB and SIMULINK*. 1st. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1996. ISBN: 0201498464 (cit. on p. 7).
- [9] S. Blumenthal, H. Bruyninckx, W. Nowak, and E. Prassler. “A scene graph based shared 3D world model for robotic applications”. In: *Proceedings of the 2013 International Conference on Robotics and Automation (ICRA 2013), Karlsruhe, Germany*. IEEE, 2013, pp. 453–460 (cit. on pp. 66, 67).
- [10] J. Bohren and S. Cousins. “The SMACH High-Level Executive”. In: *IEEE Robotics & Automation Magazine* Vol. 17.4 (Dec. 2010), pp. 18–20. ISSN: 1070-9932. DOI: 10.1109/MRA.2010.938836 (cit. on pp. 126, 130).

- [11] C. Borst et al. “A humanoid upper body system for two-handed manipulation”. In: *Proceedings of the 2007 IEEE International Conference on Robotics and Automation (ICRA 2007)*. Apr. 2007, pp. 2766–2767. DOI: 10.1109/ROBOT.2007.363886 (cit. on p. 23).
- [12] D. Brugali and P. Scandurra. “Component-Based Robotic Engineering (Part I)”. In: *IEEE Robotics & Automation Magazine* Vol. 16.4 (2009), pp. 84–96. ISSN: 1070-9932. DOI: 10.1109/MRA.2009.934837 (cit. on p. 7).
- [13] D. Brugali and A. Shakhimardanov. “Component-Based Robotic Engineering (Part II)”. In: *IEEE Robotics & Automation Magazine* Vol. 20.1 (Mar. 2010). ISSN: 1070-9932. DOI: 10.1109/MRA.2010.935798 (cit. on p. 7).
- [14] H. Bruyninckx. “Open robot control software: the OROCOS project”. In: *Proceedings of the 2001 IEEE International Conference on Robotics and Automation (ICRA 2001)*. Vol. 3. 2001, pp. 2523–2528. DOI: 10.1109/ROBOT.2001.933002 (cit. on pp. 8, 23, 66, 86, 108).
- [15] H. Bruyninckx, M. Klotzbücher, N. Hochgeschwender, G. Kraetzschmar, L. Gherardi, and D. Brugali. “The BRICS Component Model: A Model-based Development Paradigm for Complex Robotics Software Systems”. In: *Proceedings of the 28th Annual ACM Symposium on Applied Computing (SAC 2013)*. SAC '13. Coimbra, Portugal: ACM, 2013, pp. 1758–1764. ISBN: 978-1-4503-1656-9. DOI: 10.1145/2480362.2480693 (cit. on p. 108).
- [16] R. Chatila. “Mobile and Distributed Robotics”. In: *Springer Handbook of Robotics*. Springer, 2008, pp. 796–797 (cit. on p. 18).
- [17] S. Chitta, I. Sucan, and S. Cousins. “Moveit! [ROS Topics]”. In: *IEEE Robotics & Automation Magazine* Vol. 1.19 (2012), pp. 18–19 (cit. on p. 86).
- [18] P. Corke. “A robotics toolbox for MATLAB”. In: *IEEE Robotics & Automation Magazine* Vol. 3.1 (Mar. 1996), pp. 24–32. ISSN: 1070-9932. DOI: 10.1109/100.486658 (cit. on p. 7).
- [19] E. Coumans. *Bullet Physics Library*. Online, accessed Feb 2016. URL: <http://bulletphysics.org/> (cit. on p. 167).
- [20] S. Cousins. “ROS on the PR2 [ROS Topics]”. In: *IEEE Robotics & Automation Magazine* Vol. 17.3 (Sept. 2010), pp. 23–25. ISSN: 1070-9932. DOI: 10.1109/MRA.2010.938502 (cit. on p. 23).
- [21] T. De Laet, H. Bruyninckx, and J. De Schutter. “Rigid body pose and twist scene graph founded on geometric relations semantics for robotic applications”. In: *Proceedings of the 2013 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2013)*. Nov. 2013, pp. 2398–2405. DOI: 10.1109/IROS.2013.6696693 (cit. on p. 66).

- 
- [22] T. De Laet, S. Bellens, R. Smits, E. Aertbelien, H. Bruyninckx, and J. De Schutter. “Geometric Relations between Rigid Bodies (Part 1) – Semantics for Standardization”. In: *IEEE Robotics & Automation Magazine* (June 2013), pp. 84–93 (cit. on pp. 65, 66).
- [23] J. De Schutter, T. De Laet, J. Rutgeerts, W. Decré, R. Smits, E. Aertbelien, K. Claes, and H. Bruyninckx. “Constraint-based task specification and estimation for sensor-based robot systems in the presence of geometric uncertainty”. In: *The International Journal of Robotics Research* Vol. 26.5 (2007), pp. 433–455 (cit. on pp. 86, 149).
- [24] M. Dogar, R. A. Knepper, A. Spielberg, C. Choi, H. I. Christensen, and D. Rus. “Multi-scale assembly with robot teams”. In: *The International Journal of Robotics Research* Vol. 34.13 (2015), pp. 1645–1659. DOI: 10.1177/0278364915586606 (cit. on p. 30).
- [25] *DX9 Black Edition System w/ AR9020 Receiver – Spektrum*. Online, accessed Feb 2016. URL: <http://www.spektrumrc.com/Products/Default.aspx?ProdId=SPM9900> (cit. on p. 33).
- [26] J. Faust, V. Pradeep, and D. Thomas. *message\_filters*. Online, accessed Feb 2016. URL: [http://wiki.ros.org/message\\_filters](http://wiki.ros.org/message_filters) (cit. on p. 148).
- [27] R. Featherstone. *Rigid Body Dynamics Algorithms*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2007. ISBN: 0387743146 (cit. on p. 161).
- [28] E. Fernandez, E. Marder-Eppstein, and V. Pradeep. *actionlib*. Online, accessed Feb 2016. URL: <http://wiki.ros.org/actionlib> (cit. on pp. 107, 130).
- [29] T. Foote. “tf: The transform library”. In: *Proceedings of the 2013 IEEE International Conference on Technologies for Practical Robot Applications (TePRA 2013)*. Apr. 2013, pp. 1–6. DOI: 10.1109/TePRA.2013.6556373 (cit. on p. 66).
- [30] T. Foote and S. Glaser. *trajectory\_msgs*. Online, accessed Feb 2016. URL: [http://wiki.ros.org/trajectory\\_msgs](http://wiki.ros.org/trajectory_msgs) (cit. on p. 85).
- [31] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: Elements of reusable object-oriented software*. Addison Wesley, 1994 (cit. on pp. 16, 55, 90, 94, 126, 152, 153).
- [32] Y. Gan, X. Dai, and D. Li. “Off-Line Programming Techniques for Multirobot Cooperation System”. In: *International Journal of Advanced Robotic Systems* (2013). DOI: 10.5772/56506 (cit. on p. 130).
- [33] *Gazebo – Robot simulation made easy*. Online, accessed Feb 2016. URL: <http://gazebo-sim.org/> (cit. on p. 167).
- [34] B. Gerkey. *Why ROS 2.0*. Online, accessed Feb 2016. URL: [http://design.ros2.org/articles/why%5C\\_ros2.html](http://design.ros2.org/articles/why%5C_ros2.html) (cit. on p. 167).

- [35] P. Gerum. *Xenomai – Implementing a RTOS emulation framework on GNU/Linux*. Online, accessed Feb 2016. 2004. URL: <http://www.xenomai.org/documentation/xenomai-2.5/pdf/xenomai> (cit. on pp. 10, 21).
- [36] D. Harel. “Statecharts: A visual formalism for complex systems”. In: *Science of computer programming* Vol. 8.3 (1987), pp. 231–274 (cit. on p. 126).
- [37] M. Hermann, T. Pentek, and B. Otto. *Design Principles for Industrie 4.0 Scenarios: A Literature Review*. Tech. rep. Online, accessed Feb 2016. Technische Universität Dortmund, 2015. URL: [http://www.snom.mb.tu-dortmund.de/cms/de/forschung/Arbeitsberichte/Design-Principles-for-Industrie-4\\_0-Scenarios.pdf](http://www.snom.mb.tu-dortmund.de/cms/de/forschung/Arbeitsberichte/Design-Principles-for-Industrie-4_0-Scenarios.pdf) (cit. on p. 2).
- [38] D. Hildebrand. “An Architectural Overview of QNX.” In: *USENIX Workshop on Microkernels and Other Kernel Architectures*. 1992, pp. 113–126 (cit. on p. 21).
- [39] A. Hoffmann. “Serviceorientierte Automatisierung von Roboterzellen”. PhD thesis. University of Augsburg, Apr. 2015 (cit. on pp. 1, 2, 8, 10, 17–19, 55, 65, 112, 126, 129, 130, 181).
- [40] A. Hoffmann, A. Angerer, F. Ortmeier, M. Vistein, and W. Reif. “Hiding Real-Time: A new Approach for the Software Development of Industrial Robots”. In: *Proceedings of the 2009 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2009)*, St. Louis, Missouri, USA. IEEE, Oct. 2009, pp. 2108–2113 (cit. on pp. 8, 9, 23).
- [41] N. Hogan. “Impedance control: An approach to manipulation: Part I - theory”. In: *Journal of dynamic systems, measurement, and control* Vol. 107.1 (1985), pp. 1–7 (cit. on p. 77).
- [42] Hokuyo Automatic Co. Ltd. *Communication Protocol Specification For SCIP2.0 Standard*. Online, accessed Feb 2016. URL: [http://www.hokuyo-aut.jp/02sensor/07scanner/download/pdf/URG\\_SCIP20.pdf](http://www.hokuyo-aut.jp/02sensor/07scanner/download/pdf/URG_SCIP20.pdf) (cit. on p. 163).
- [43] Hokuyo Automatic Co. Ltd. *Scanning range finder URG-04LX-UG01*. Online, accessed Feb 2016. URL: [https://www.hokuyo-aut.jp/02sensor/07scanner/urg\\_04lx\\_ug01.html](https://www.hokuyo-aut.jp/02sensor/07scanner/urg_04lx_ug01.html) (cit. on p. 33).
- [44] B. E. Ilon. “Wheels For A Course Stable Selfpropelling Vehicle Movable In Any Desired Direction On The Ground Or Some Other Base”. US3876255. Apr. 8, 1975 (cit. on p. 30).
- [45] International Federation of Robotics. *World Robotics 2015*. Online, accessed Feb 2016. URL: [http://www.worldrobotics.org/uploads/tx\\_zeifr/Executive\\_Summary\\_WR\\_2015.pdf](http://www.worldrobotics.org/uploads/tx_zeifr/Executive_Summary_WR_2015.pdf) (cit. on pp. 1, 2).

- 
- [46] T. Jacobs. *Standardisation Efforts on Industrial and Service Robots*. Online, accessed Feb 2016. June 2015. URL: [https://eu-robotics.net/cms/upload/downloads/ISO-Standardisation-Newsletter\\_2015-06.pdf](https://eu-robotics.net/cms/upload/downloads/ISO-Standardisation-Newsletter_2015-06.pdf) (cit. on p. 184).
- [47] C. Jang, S.-I. Lee, S.-W. Jung, B. Song, R. Kim, S. Kim, and C.-H. Lee. “OPRoS: A new component-based robot software platform”. In: *Electronics and Telecommunications Research Institute Journal* Vol. 32.5 (2010), pp. 646–656 (cit. on p. 8).
- [48] D. Jansen and H. Buttner. “Real-time Ethernet: the EtherCAT solution”. In: *Computing & Control Engineering Journal* Vol. 15 (1 Feb. 2004), pp. 16–21. ISSN: 0956-3385 (cit. on pp. 23, 31, 160).
- [49] *JBullet – Java port of Bullet Physics Library*. Online, accessed Feb 2016. URL: <http://jbullet.advel.cz/> (cit. on p. 167).
- [50] E. G. Jones. *arm\_navigation*. Online, accessed Feb 2016. URL: [http://wiki.ros.org/arm\\_navigation](http://wiki.ros.org/arm_navigation) (cit. on p. 86).
- [51] S. Julier, J. Uhlmann, and H. Durrant-Whyte. “A new approach for filtering nonlinear systems”. In: *Proceedings of the 1995 American Control Conference*. Vol. 3. June 1995, pp. 1628–1632. DOI: 10.1109/ACC.1995.529783 (cit. on p. 148).
- [52] H. Kagermann, J. Helbig, A. Hellinger, and W. Wahlster. *Recommendations for Implementing the Strategic Initiative INDUSTRIE 4.0: Securing the Future of German Manufacturing Industry; Final Report of the Industrie 4.0 Working Group*. Online, accessed Feb 2016. Forschungsunion, 2013. URL: [http://www.acatech.de/fileadmin/user\\_upload/Baumstruktur\\_nach\\_Website/Acatech/root/de/Material\\_fuer\\_Sonderseiten/Industrie\\_4.0/Final\\_report\\_\\_Industrie\\_4.0\\_accessible.pdf](http://www.acatech.de/fileadmin/user_upload/Baumstruktur_nach_Website/Acatech/root/de/Material_fuer_Sonderseiten/Industrie_4.0/Final_report__Industrie_4.0_accessible.pdf) (cit. on p. 1).
- [53] R. E. Kalman. “A new approach to linear filtering and prediction problems”. In: *Journal of Fluids Engineering* Vol. 82.1 (1960), pp. 35–45 (cit. on p. 148).
- [54] B. Keiser. “Torque Control of a KUKA youBot Arm”. Online, accessed Feb 2016. MA thesis. University of Zurich, 2013. URL: [http://rpg.ifi.uzh.ch/docs/theses/Benjamin\\_Keiser\\_Torque\\_Control\\_2013.pdf](http://rpg.ifi.uzh.ch/docs/theses/Benjamin_Keiser_Torque_Control_2013.pdf) (cit. on pp. 30, 167).
- [55] J. Kelly. *LEGO MINDSTORMS NXT-G Programming Guide*. Technology in Action. Apress, 2010. ISBN: 9781430229773 (cit. on p. 7).
- [56] O. Khatib, K. Yokoi, K. Chang, D. Ruspini, R. Holmberg, and A. Casal. “Coordination and decentralized cooperation of multiple mobile manipulators”. In: *Journal of Robotic Systems* Vol. 13.11 (1996), pp. 755–764. ISSN: 1097-4563. DOI: 10.1002/(SICI)1097-4563(199611)13:11<755::AID-ROB6>3.0.CO;2-U (cit. on p. 30).

- [57] M. Klotzbücher, G. Biggs, and H. Bruyninckx. “Pure coordination using the Coordinator–Configurator Pattern”. In: *Proceedings of the 3rd International Workshop on Domain-Specific Languages and models for ROBotic systems (DSLRob-12)*. Vol. abs/1303.0066. 2013 (cit. on p. 23).
- [58] M. Klotzbucher and H. Bruyninckx. “Coordinating Robotic Tasks and Systems with rFSM Statecharts”. In: *Journal of Software Engineering for Robotics* Vol. 3.1 (2012), pp. 28–56 (cit. on pp. 108, 109, 113, 126, 131).
- [59] R. A. Knepper, T. Layton, J. Romanishin, and D. Rus. “IkeaBot: An autonomous multi-robot coordinated furniture assembly system”. In: *Proceedings of the 2013 IEEE International Conference on Robotics and Automation (ICRA)*. May 2013, pp. 855–862. DOI: 10.1109/ICRA.2013.6630673 (cit. on p. 30).
- [60] N. Koenig and A. Howard. “Design and use paradigms for Gazebo, an open-source multi-robot simulator”. In: *Proceedings of the 2004 IEEE/RSJ International Conference on Intelligent Robots and Systems, 2004. (IROS 2004)*. Vol. 3. Sept. 2004, pp. 2149–2154. DOI: 10.1109/IROS.2004.1389727 (cit. on p. 167).
- [61] KUKA Roboter GmbH. *KUKA youBot – Research & Application Development in Mobile Robotics*. Online, accessed Feb 2016. URL: [http://www.kuka-robotics.com/res/sps/9cb8e311-bfd7-44b4-b0ea-b25ca883f6d3\\_youBot\\_data\\_sheet.pdf](http://www.kuka-robotics.com/res/sps/9cb8e311-bfd7-44b4-b0ea-b25ca883f6d3_youBot_data_sheet.pdf) (cit. on pp. 30–32).
- [62] *KUKA youBot API*. Online, accessed Feb 2016. URL: <http://www.youbot-store.com/developers/software/libraries/kuka-youbot-api> (cit. on p. 31).
- [63] *KUKA youBot kinematics, dynamics and 3D model*. Online, accessed Feb 2016. URL: <http://www.youbot-store.com/developers/documentation/kuka-youbot-kinematics-dynamics-and-3d-model> (cit. on p. 161).
- [64] K.-K. Lau, L. Ling, and P. Velasco Elizondo. “Towards Composing Software Components in Both Design and Deployment Phases”. In: *Proceedings of the 10th International Symposium on Component-Based Software Engineering (CBSE 2007), Medford, MA, USA, July 9-11, 2007*. Ed. by H. W. Schmidt, I. Crnkovic, G. T. Heineman, and J. A. Stafford. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 274–282. ISBN: 978-3-540-73551-9 (cit. on p. 152).
- [65] M. S. Loffler, V. Chitrakaran, and D. M. Dawson. “Design and Implementation of the Robotic Platform”. In: *Journal of Intelligent and Robotic System* Vol. 39 (2004), pp. 105–129 (cit. on p. 8).
- [66] D. Lu. *URDF and You*. Online, accessed Feb 2016. May 2012. URL: <http://wustl.probablydavid.com/publications/URDFandYou.pdf> (cit. on p. 67).



- 
- [67] N. Mansard, O. Stasse, P. Evrard, and A. Kheddar. “A versatile Generalized Inverted Kinematics implementation for collaborative working humanoid robots: The Stack Of Tasks”. In: *Proceedings of the 2009 International Conference on Advanced Robotics (ICAR 2009)*. June 2009, pp. 1–6 (cit. on p. 86).
- [68] P. Mantegazza, L. Dozio, and S. Papacharalambous. “RTAI: Real time application interface”. In: *Linux Journal* Vol. 72.10 (2000) (cit. on p. 21).
- [69] E. Marder-Eppstein, D. V. Lu, and M. Ferguson. *move\_base*. Online, accessed Feb 2016. URL: [http://wiki.ros.org/move\\_base](http://wiki.ros.org/move_base) (cit. on p. 86).
- [70] M. J. Matarić and F. Michaud. “Behavior-Based Systems”. In: *Springer Handbook of Robotics*. Springer, 2008, pp. 891–906 (cit. on p. 19).
- [71] W. Meeussen. *Coordinate Frames for Mobile Platforms*. Online, accessed Feb 2016. Oct. 2010. URL: <http://www.ros.org/repos/rep-0105.html> (cit. on p. 148).
- [72] W. Meeussen and D. V. Lu. *robot\_pose\_ekf*. Online, accessed Feb 2016. URL: [http://wiki.ros.org/robot\\_pose\\_ekf](http://wiki.ros.org/robot_pose_ekf) (cit. on p. 149).
- [73] T. Moore and D. Stouch. “A Generalized Extended Kalman Filter Implementation for the Robot Operating System”. In: *Proceedings of the 13th International Conference on Intelligent Autonomous Systems (IAS-13)*. Springer, July 2014 (cit. on p. 149).
- [74] S. S. Muchnick. *Advanced Compiler Design and Implementation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997. ISBN: 1-55860-320-4 (cit. on p. 92).
- [75] E. Mueggler, M. Faessler, F. Fontana, and D. Scaramuzza. “Aerial-guided navigation of a ground robot among movable obstacles”. In: *Proceedings of the 2014 IEEE International Symposium on Safety, Security, and Rescue Robotics (SSRR)*. Oct. 2014, pp. 1–8. DOI: 10.1109/SSRR.2014.7017662 (cit. on p. 30).
- [76] H. Mühe, A. Angerer, A. Hoffmann, and W. Reif. “On reverse-engineering the KUKA Robot Language”. In: *1st International Workshop on Domain-Specific Languages and models for ROBotic systems (DSLRob ’10), 2010 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2010)* (2010) (cit. on p. 7).
- [77] H. Neugass, G. Espin, H. Nunoe, R. Thomas, and D. Wilner. “VxWorks: An interactive development environment and real-time kernel for GMicro”. In: *Proceedings of The Eighth TRON Project Symposium (International)*. 1991, pp. 196–207 (cit. on p. 21).
- [78] L. E. Parker. “Multiple Mobile Robot Systems”. In: *Springer Handbook of Robotics*. Springer, 2008, pp. 921–941 (cit. on p. 19).
- [79] C. Pelich and F. M. Wahl. “ZERO++: An OOP Environment for Multiprocessor Robot Control”. In: *International Journal of Robotics & Automation* Vol. 12.2 (1997), pp. 49–57 (cit. on p. 8).

- [80] T. Pietzsch. *rOsewhite Saphira*. Online, accessed Feb 2016. URL: <http://www.rosewhite.de/saphira> (cit. on p. 32).
- [81] H. Putnam. “Three-valued logic”. In: *Philosophical Studies* Vol. 8.5 (1957), pp. 73–80 (cit. on p. 12).
- [82] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng. “ROS: an open-source Robot Operating System”. In: *Workshop on Open Source Software, 2009 IEEE International Conference on Robotics & Automation (ICRA 2009)*. 2009 (cit. on pp. 8, 25, 66, 107).
- [83] M. Radestock and S. Eisenbach. “Coordination in Evolving Systems”. In: *Proceedings of the International Workshop on Trends in Distributed Systems: CORBA and Beyond*. TreDS ’96. London, UK: Springer-Verlag, 1996, pp. 162–176. ISBN: 3-540-61842-2 (cit. on p. 113).
- [84] B. Raphael. *Robot Research at Stanford Research Institute*. Tech. rep. Stanford Research Institute, Feb. 1972 (cit. on p. 2).
- [85] *ROS Wrapper for KUKA youBot API*. Online, accessed Feb 2016. URL: <http://www.youbot-store.com/developers/software/ros/ros-wrapper-for-kuka-youbot-api> (cit. on p. 31).
- [86] M. C. Ruiz. “Haptic Teleoperation of the youBot with friction compensation for the base”. MA thesis. Departamento de Ingeniería de Sistema y Automática, Universidad Carlos III de Madrid, 2012. URL: <http://hdl.handle.net/10016/16267> (cit. on p. 168).
- [87] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. E. Lorensen, et al. *Object-oriented modeling and design*. Vol. 199. 1. Prentice-hall Englewood Cliffs, 1991 (cit. on pp. 8, 67).
- [88] A. Schierl, A. Angerer, A. Hoffmann, M. Vistein, and W. Reif. “A Taxonomy of Distribution for Cooperative Mobile Manipulators”. In: *Proceedings of the 12th International Conference on Informatics in Control, Automation & Robotics (ICINCO 2015), Rome, Italy*. 2015 (cit. on pp. 19, 20, 156).
- [89] A. Schierl, A. Angerer, A. Hoffmann, M. Vistein, and W. Reif. “From Robot Commands To Real-Time Robot Control - Transforming High-Level Robot Commands into Real-Time Dataflow Graphs”. In: *Proceedings of the 9th International Conference on Informatics in Control, Automation and Robotics (ICINCO 2012), Rome, Italy*. 2012 (cit. on pp. 16, 65, 85, 96, 106, 107).

- 
- [90] A. Schierl, A. Angerer, A. Hoffmann, M. Vistein, and W. Reif. “On Structure and Distribution of Software for Mobile Manipulators”. In: *Informatics in Control, Automation and Robotics: 12th International Conference, ICINCO 2015 Colmar, Alsace, France, 21-23 July, 2015 Revised Selected Papers*. Ed. by J. Filipe, O. Gusikhin, K. Madani, and J. Sasiadek. Springer International Publishing, 2016 (cit. on p. 19).
- [91] A. Schierl, A. Hoffmann, A. Angerer, M. Vistein, and W. Reif. “Towards Realtime Robot Reactions: Patterns for Modular Device Driver Interfaces”. In: *Workshop on Software Development & Integration (SDIR 2013), IEEE International Conference on Robotics & Automation (ICRA 2013), Karlsruhe, Germany*. 2013 (cit. on p. 22).
- [92] T. Schneider. “Distributed Networks Using ROS – Cross-Network Middleware Communication Using IPv6”. MA thesis. Technische Universität München, Nov. 2012 (cit. on p. 167).
- [93] E. Scioni, M. Klotzbuecher, T. De Laet, H. Bruyninckx, and M. Bonfe. “Preview coordination: An enhanced execution model for online scheduling of mobile manipulation tasks”. In: *Proceedings of the 2013 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2013)*. Nov. 2013, pp. 5779–5786. DOI: 10.1109/IROS.2013.6697193 (cit. on p. 131).
- [94] S. Sharma, G. K. Kraetzschmar, C. Scheurer, and R. Bischoff. “Unified Closed Form Inverse Kinematics for the KUKA youBot”. In: *Proceedings of 7th German Conference on Robotics (ROBOTIK 2012)*. May 2012, pp. 1–6 (cit. on p. 87).
- [95] R. Smits. *Orocos Kinematics and Dynamics*. Online, accessed Feb 2016. URL: <http://www.orocos.org/kdl> (cit. on pp. 66, 161).
- [96] *State Chart XML (SCXML): State Machine Notation for Control Abstraction*. Working Draft 6. Online, accessed Feb 2016. W3C, Dec. 2012. URL: <http://www.w3.org/TR/scxml/> (cit. on p. 126).
- [97] I. A. Sutan and S. Chitta. *MoveIt!* Online, accessed Feb 2016. URL: <http://moveit.ros.org> (cit. on p. 86).
- [98] D. Theunissen. *Tx1-K1 – Deltang*. Online, accessed Feb 2016. URL: <http://www.deltang.co.uk/tx1k1-101.htm> (cit. on p. 33).
- [99] G. Tonietti, R. Schiavi, and A. Bicchi. “Design and Control of a Variable Stiffness Actuator for Safe and Fast Physical Human/Robot Interaction”. In: *Proceedings of the 2005 IEEE International Conference on Robotics and Automation (ICRA 2005)*. Apr. 2005, pp. 526–531. DOI: 10.1109/ROBOT.2005.1570172 (cit. on p. 105).

- [100] A. Tsiamis, C. K. Verginis, C. P. Bechlioulis, and K. J. Kyriakopoulos. “Cooperative manipulation exploiting only implicit communication”. In: *Proceedings of the 2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. Sept. 2015, pp. 864–869. DOI: 10.1109/IROS.2015.7353473 (cit. on p. 30).
- [101] D. Vanthienen, M. Klotzbücher, and H. Bruyninckx. “The 5C-based architectural Composition Pattern: lessons learned from re-developing the iTaSC framework for constraint-based robot programming”. In: *Journal of Software Engineering for Robotics* Vol. 5.1 (2014), pp. 17–35 (cit. on pp. 86, 109).
- [102] Vicon Motion Systems Ltd. *Go Further With Vicon MX T-Series*. Online, accessed Feb 2016. URL: <http://www.vicon.com/downloads/documentation/vicon-documentation/t-series-go-further-v13> (cit. on p. 35).
- [103] M. Vistein. “Embedding Real-Time Critical Robotics Applications in an Object-Oriented Language”. PhD thesis. University of Augsburg, Mar. 2015 (cit. on pp. 1, 2, 8, 10, 15, 18, 23, 96, 97, 106, 107, 160, 164, 166, 181–183).
- [104] K. Vollmann. “A new approach to robot simulation tools with parametric components”. In: *Proceedings of the 2002 IEEE International Conference on Industrial Technology (ICIT ’02)*. Vol. 2. Dec. 2002, pp. 881–885. DOI: 10.1109/ICIT.2002.1189284 (cit. on p. 7).
- [105] K. Waldron and J. Schmiedeler. “Kinematics”. In: *Springer Handbook of Robotics*. Springer, 2008, pp. 9–33 (cit. on pp. 41, 42).
- [106] R. Wang and X. Qian. *OpenSceneGraph 3.0: Beginner’s guide*. Packt Publishing Ltd, 2010 (cit. on p. 67).
- [107] C. Zieliński. “Object-oriented robot programming”. In: *Robotica* Vol. 15.1 (1997), pp. 41–48 (cit. on p. 8).

## LIST OF FIGURES

1.1	Structure of this work . . . . .	4
2.1	Software architecture separating applications from real-time concerns . . . . .	10
2.2	Structure of the Robot Control Core . . . . .	11
2.3	Example data-flow graph for a robot joint motion . . . . .	13
2.4	Example synchronization rules for motion sequences . . . . .	14
2.5	Example synchronization rules for device concurrency . . . . .	15
2.6	Software structure for distributed robots with real-time guarantees . . . . .	20
2.7	Examples of software structure on the real-time level . . . . .	21
2.8	Examples of software distribution on the real-time level . . . . .	22
2.9	Examples of software structure on the system level . . . . .	24
2.10	Examples of software distribution on the system level . . . . .	24
2.11	Examples of software structure on the application level . . . . .	26
2.12	Examples of software distribution on the application level . . . . .	27
3.1	The KUKA youBot mobile manipulator . . . . .	31
3.2	The r0sewhite Saphira quadcopter . . . . .	33
3.3	The sensors used in the case study . . . . .	34
3.4	Application example: Handover in motion . . . . .	35
3.5	Application example: Gesture control of the quadcopter . . . . .	37
4.1	Primitive types in Cartesian space . . . . .	43
4.2	Example set-up with four Frames and their Relations . . . . .	44
4.3	Frames and Relations describing geometry . . . . .	45
4.4	Example set-up with different Points and Orientations in Cartesian space . . . . .	47
4.5	Software model for Points and Orientations . . . . .	48
4.6	Poses and FramePoses in Cartesian space . . . . .	49
4.7	Example set-up with different Frames, Poses and FramePoses . . . . .	50
4.8	Example for different ways of expressing the velocity of a rotating Frame . . . . .	52
4.9	Velocities in Cartesian space . . . . .	53
4.10	A rigid body and its object-oriented representation . . . . .	54

4.11 Concepts related to PhysicalObjects . . . . .	55
4.12 The youBot platform modeled as a physical object . . . . .	57
4.13 The youBot arm modeled as a physical object . . . . .	58
4.14 The youBot gripper modeled as physical objects . . . . .	59
4.15 The youBot soft fingers and their closed kinematic chain . . . . .	60
4.16 The Saphira Quadcopter modeled as a physical object . . . . .	61
4.17 Hokuyo laser scanner modeled as a physical object . . . . .	61
4.18 Entity and geometry definition for a youBot platform as a mobile manipulator . . .	62
4.19 Application geometry for the handover scenario . . . . .	63
4.20 Application geometry for the gesture control scenario . . . . .	64
5.1 Cartesian space motions . . . . .	72
5.2 Class diagram of available Cartesian space motions . . . . .	73
5.3 Joint space motions . . . . .	74
5.4 Class diagram of available Joint space motions . . . . .	75
5.5 Impedance control in Cartesian space . . . . .	77
5.6 World model for a youBot platform executing a linear motion . . . . .	79
5.7 Example manipulator with position and orientation preserving projector . . . . .	80
5.8 Kinematic structure and restriction of the youBot arm . . . . .	81
5.9 OrientationPreservingProjector for the youBot arm . . . . .	81
5.10 PositionPreservingProjector for the youBot arm . . . . .	82
5.11 PositionPreservingProjector for the youBot platform . . . . .	84
6.1 RealtimeValues for specifying computation laws . . . . .	91
6.2 Class diagram of concepts related to Commands . . . . .	93
6.3 Typical types of RealtimeValueSinks . . . . .	94
6.4 Class diagram of concepts related to the execution of Commands . . . . .	95
6.5 Concepts involved in Data-flow graphs . . . . .	97
6.6 Typical ActionResults for motions . . . . .	98
6.7 ActionResult for an accelerated motion of one Joint . . . . .	99
6.8 Java code to create a JointPositionActionResult for an accelerated motion . . . . .	99
6.9 Concepts involved in transforming Commands into Data-flow graphs . . . . .	100
6.10 Fragments used in the transformation . . . . .	101
6.11 Example sequence for transforming a Command into a data-flow graph . . . . .	102
6.12 Handling Command life-cycle through synchronization rules . . . . .	103
7.1 Class diagram of concepts related to Activitys and DeviceInterfaces . . . . .	116
7.2 Parallelism and error handling with beginExecute() . . . . .	118
7.3 Class diagram with concepts related to Activity execution . . . . .	119
7.4 Preparation process for beginExecute() . . . . .	120

7.5	Example control flow using <code>beginExecute()</code> for sequential execution . . . . .	122
7.6	Example control flow using <code>beginExecute()</code> for parallel execution . . . . .	123
7.7	Java code using a <code>ThreadActivity</code> . . . . .	124
7.8	Example control flow for a <code>ThreadActivity</code> . . . . .	125
7.9	Classes used with <code>StateChartActivitys</code> . . . . .	127
8.1	Specialized <code>RealtimeValueListeners</code> for handling sensor data . . . . .	135
8.2	Timing behavior of different sensors . . . . .	136
8.3	Java code combining sensor values . . . . .	139
8.4	Observations used to integrate sensor data into the World model . . . . .	141
8.5	Example World model with Relations and Observations . . . . .	142
8.6	Geometry definition for a youBot with Vicon position tracking . . . . .	143
8.7	Effects of the Estimator for a youBot platform tracked using Vicon tracking . . . . .	144
9.1	Devices and <code>RoboticsRuntimes</code> . . . . .	153
9.2	Adding and modifying <code>DeviceInterfaces</code> . . . . .	154
9.3	Example for <code>DeviceInterfaces</code> for robot arms . . . . .	155
9.4	Example deployment when controlling two youBots with Vicon tracking . . . . .	157
9.5	Excerpt of classes used with the Java Control Core . . . . .	159
9.6	Position control loop with velocity feed-forward for the youBot . . . . .	161
9.7	Cascaded control loop for the quadcopter . . . . .	162
9.8	Different ways of handling time delays introduced by wireless communication . . . . .	164
9.9	Data points and approximations used in the proposed time-smoothing process . . . . .	165
10.1	Workflow for picking up the baton using parallelism with <code>beginExecute()</code> . . . . .	170
10.2	Workflow for passing the baton using synchronization with <code>beginExecute()</code> . . . . .	171
10.3	Handover workflow with state charts for reduced time delays . . . . .	172
10.4	Java code to create and execute the <code>StateChartActivity</code> for picking up the baton . . . . .	173
10.5	Workflow state chart for the gesture control application . . . . .	175
10.6	Example application: Gesture control . . . . .	176
10.7	Java code to determine the wand activation . . . . .	177





## INDEX

- Action, 70
  - Arm, 76
  - CartesianErrorCorrection, 73
  - CartesianJogging, 74
  - CartesianSuperposition, 73
  - CartesianTrajectory, 72
  - CartesianTrajectoryFromMotion, 73
  - CIRC, 72
  - Completed, 71
  - Disarm, 76
  - FollowJointGoal, 74
  - FollowJointVelocity, 75
  - FollowPose, 72
  - FollowVelocity, 74
  - Goal Action, 70
  - HoldJointPosition, 75
  - HoldJointVelocity, 75
  - HoldPose, 73
  - HoldVelocity, 74
  - JointErrorCorrection, 75
  - JointJogging, 75
  - JointSpline, 74
  - JointSuperposition, 75
  - JointTrajectory, 74
  - JointTrajectoryFromMotion, 75
  - LIN, 72
  - MoveGripper, 75
  - MoveToJointGoal, 74
  - MoveToPose, 72
  - Process Action, 70
  - PTP, 74
  - SPLINE, 72
  - SwitchToJointImpedance, 76
  - SwitchToPositionControl, 76
- ActionMapper, 100
- ActionResult, 98
  - CartesianGoalActionResult, 98
  - CartesianVelocityActionResult, 98
  - JointGoalActionResult, 98
  - JointPositionActionResult, 98
  - JointVelocityActionResult, 98
  - MultiJointGoalActionResult, 98
  - MultiJointPositionActionResult, 98
  - MultiJointVelocityActionResult, 98
- Activity, 115
  - beginExecute(), 116
  - prepareForResult(), 119
  - cancelExecute(), 116
  - endExecute(), 116
  - execute(), 116
  - ConditionalActivity, 124
  - ParallelActivity, 123
  - SequentialActivity, 122
  - StateChartActivity, 126
  - ThreadActivity, 124
- ActivityResult, 118
  - done, 119
  - failed, 119
- ActivityResultListener, 120
- ActivitySchedule, 119
- Actuator, 70
- ActuatorDriver, 153

- ActuatorDriverMapper, 100
- ActuatorInterface, 115
- AddedRealtimeDouble, 91
- Arm, 76
- Assignment, 93
- BinaryFunctionRealtimeDouble, 91
- Cancel, 95
- CartesianErrorCorrection, 73
- CartesianGoalActionResult, 98
- CartesianGoalMotionInterface, 115
- CartesianJogging, 74
- CartesianPathMotionInterface, 115
- CartesianPositionActionResult, 98
- CartesianSuperposition, 73
- CartesianTrajectory, 72
- CartesianTrajectoryFromMotion, 73
- CartesianVelocityActionResult, 98
- CartesianVelocityMotionInterface, 115
- CIRC, 72
- Command, 90
  - addAssignment(), 93
  - addCancelCondition(), 93
  - addCompletionResult(), 93
  - addExceptionCondition(), 94
  - addExceptionHandler(), 94
  - addTakeoverResult(), 93
  - ignoreException(), 94
  - load(), 94
- CommandHandle, 94
  - abort(), 95
  - cancel(), 95
  - start(), 94
  - waitComplete(), 95
- CommandHandleOperation, 95
- CommandRealtimeException, 94
- CommandResult, 93
- Condition, 126
- ConditionalActivity, 124
- ConstantRealtimeDouble, 91
- ConstantRealtimeInteger, 91
- Data synchronization, 113
- Data-flow graph, 11
- Device, 70
  - use(), 115
- Device driver, 11
- DeviceDriver, 153
- DeviceInterface, 115
- DeviceInterfaceFactory, 154
- DeviceInterfaceListener, 154
- DeviceInterfaceModifier, 154
- DeviceParameter, 76
  - CartesianLimit, 76
  - FrameProjector, 80
  - JointImpedanceParameter, 78
  - JointLimit, 76
  - MotionCenter, 76
  - QuadcopterAttitudeController, 78
  - QuadcopterPositionController, 78
- Disarm, 76
- DropSensorListener, 135
- DynamicConnection, 56
- Entity, 55
- Estimator, 143
- execution environment, 90
- FollowJointGoal, 74
- FollowJointVelocity, 75
- FollowPose, 72
- FollowVelocity, 74
- Fragment, 11
- FragmentInPort, 96
- FragmentOutPort, 96
- Frame, 44
- FrameObservation, 141
- FramePose, 50

- asPose(), 51
- withFrame(), 50
- withReferenceAndFrame(), 51
- FramePoseObservation, 141
- FrameProjector, 76
  - OrientationPreservingProjector, 80
  - PositionPreservingProjector, 80
- FrameTopology, 46
- FrameVelocity, 54
  - withFrame(), 54
- Goal Action, 70
- HoldJointPosition, 75
- HoldJointVelocity, 75
- HoldPose, 73
- HoldVelocity, 74
- InPort, 96
- Java Control Core, 158
- JavaRuntime, 159
- JDevice, 158
- Joint, 56
- JointErrorCorrection, 75
- JointGoalActionResult, 98
- JointImpedanceController, 78
- JointJogging, 75
- JointPositionActionResult, 98
- JointSpline, 74
- JointSuperposition, 75
- JointTrajectory, 74
- JointTrajectoryFromMotion, 75
- JointVelocityActionResult, 98
- JPrimitive, 158
- LIN, 72
- Link, 56
- Mapper, 100
- MapperRegistry, 100
- MoveGripper, 75
- MoveToJointGoal, 74
- MoveToPose, 72
- MultiJointGoalActionResult, 98
- MultiJointPositionActionResult, 98
- MultiJointVelocityActionResult, 98
- MultipliedRealtimeDouble, 91
- Observation, 140
  - FrameObservation, 141
  - FramePoseObservation, 141
  - PoseObservationFromApplication, 141
  - ViconObservation, 141
- Orientation, 47
- OrientationPreservingProjector, 80
- OutPort, 96
- ParallelActivity, 123
- Parameter, 96
- ParameterAssignment, 93
- PersistContext, 94
- PhysicalObject, 55
- Pivot point, 52
- Placement, 62
- Point, 47
- Pose, 48
  - asFramePose(), 51
  - withOrientation(), 49
  - withReference(), 49
- PoseObservationFromApplication, 141
- PoseObservationFromRealtimeValue, 141
- PositionController, 78
- PositionPreservingProjector, 80
- PreviousActivityExecutionFailedException, 117
- Primitive, 11
- Process Action, 70
- Property, 118
- PTP, 74
- QueueSensorListener, 135

- Realtime Primitives Interface, 96
- RealtimeBoolean, 91
- RealtimeDouble, 91
- RealtimeDoubleArray, 91
- RealtimeDoubleFromApplication, 92
- RealtimeIntegerFromApplication, 92
- RealtimePoint, 91
- RealtimePose, 91
- RealtimeTransformation, 91
- RealtimeTuple, 138
- RealtimeTwist, 91
- RealtimeValue, 90
  - forAge(), 137
  - fromTime(), 137
  - getConsistentAge(), 137
  - getCurrentValue(), 134
  - getDataAge(), 136
  - getTimeForAge(), 137
- RealtimeValueConsumerFragment, 100
- RealtimeValueFragment, 101
- RealtimeValueListener, 93
- RealtimeValueMapper, 101
- RealtimeValueReporting, 93
- RealtimeValueSink, 93
- RealtimeVector, 91
- RealtimeVelocity, 91
- Relation, 44
- Robot Control Core, 10, 160
- RoboticsRuntime, 94
  - scheduleOperations(), 96
- Rotation, 43
- Sensor, 134
- SensorInterface, 115
- SEntity, 159
- SequentialActivity, 122
- SimpleEstimator, 144
- SoftRobotRuntime, 160
- SPLINE, 72
- State, 126
- StateChartActivity, 126
- StaticConnection, 56
- SwitchToJointImpedance, 76
- SwitchToPositionControl, 76
- SWorld, 159
- Synchronization
  - Data synchronization, 113
  - Time synchronization, 113
  - Workflow synchronization, 113
- Synchronization rule, 14
  - Cancel Nets, 14
  - Start Nets, 14
  - Stop Nets, 14
  - Synchronization Condition, 14
- Takeover, 116
- ThreadActivity, 124
- Time synchronization, 113
- TimeAwareEstimator, 144
- Transformation, 43
- Transition, 126
- Twist, 43
- UnknownPlacement, 62
- Vector, 43
- Velocity, 51
  - withOrientation(), 53
  - withPivot(), 54
  - withReference(), 52
- ViconObservation, 141
- ViconSubject, 62
- Workflow synchronization, 113
- World model, 46